Towards Developing Effective Fact Extractors

Greg Chase, Jyotsna Das and Scott Davis

Command and Control Division
Defence Science and Technology Organisation

DSTO-TR-1729

ABSTRACT

DSTO has a program of research into automated text processing. Part of this research has led to the development of a prototype information extraction system known as the DSTO Fact Extractor System. This system can be used to extract interesting information from free text documents. Part of applying the DSTO technology involves a skilled user developing a set of one or more fact extractors that control the behaviour of the information extraction engine. These fact extractors are developed with the aid of an integrated development environment known as the Fact Extractor Workbench. This report uses a range of examples to discuss the issues that must be considered when developing fact extractors.

RELEASE LIMITATION

Approved for public release

Towards Developing Effective Fact Extractors

Executive Summary

The Australian Defence Intelligence Community (ADIC) has access to large volumes of unstructured text that require processing. The quantity of text far exceeds their capacity to process this information in a timely manner. DSTO has responded with a program of research into automated text processing. Part of this research has led to the development of a prototype information extraction system known as the DSTO Fact Extractor System.

A key feature of the DSTO Fact Extractor System is the ability to tailor its use for a particular information extraction need. An interactive development environment known as the DSTO Fact Extractor Workbench has been produced to assist the mechanics of this task but it remains somewhat of an art to create specific fact extractors.

The goal of this paper is to provide some guidance on the task of developing fact extractors that are optimised for their intended purpose. The paper introduces some commonly used fact extractor design patterns or techniques that are applicable to many information extraction tasks. The paper then shows how these design patterns may be used on real information extraction tasks by recommending a development process and working through a number of realistic case studies. The paper also discusses evaluation and the tradeoffs that may be required between precision and accuracy. The work reported in this paper was carried out under the task INT 02/290.

Authors

Greg Chase
Command and Control Division

Greg Chase graduated with honours from Flinders University in 1977. He joined the Electronic Warfare Division of the then Defence Research Centre Salisbury in 1978. He participated in the development of electronic battlefield simulations and had the technical lead in the development of an electronic warfare command and control system. He joined the Information Technology Division in 1989 where he has been involved in work on Hypermedia systems, Management of Geographic Information and more recently Information Extraction. Greg was the national leader for Australia's participation in Coalition-CINC21 (Commander IN Chief in the 21st century).

——————————————— ———————————————————————————————

Jyotsna Das
Command and Control Division

Jyotsna works in the Intelligence Analysis Group. She joined the Information Technology Division of DSTO in 1989 where she has been involved in a number of projects in areas including hypermedia systems and geospatial information management. More recently her work has focused on content extraction and text mining.

——————————————— ———————————————————————————————

Scott Davis
Command and Control Division

Scott Davis graduated with honours from the University of Adelaide in 1989. He joined the then Information Technology Division of DSTO in 1990. He has been involved in a number of projects in areas such as Management and Discovery of Geospatial Information, Hypermedia, and Information Extraction. Scott was involved in the Geospatial Data aspects of Coalition CINC-21.

——————————————— ———————————————————————————————

Contents

1. Introduction

The Australian Defence Intelligence Community (ADIC) has access to large volumes of unstructured text that require processing. The quantity of text far exceeds their capacity to process this information in a timely manner. DSTO has responded with a program of research [1] into automated text processing. Part of this research has led to the development of a prototype information extraction system known as the DSTO Fact Extractor System.

Information extraction is the process of using computers to recover information from texts in a variety of formats - such as ASCII files, emails, web pages - and convert it into a more usable form. Unlike information retrieval, which is concerned with finding whole documents to present to a user, information extraction deals with information at a much finer grain. The extracted information can either be presented directly to a human or can be passed on to a downstream process for consolidation and further processing.

The DSTO Fact Extractor System is a text-processing framework that uses software agents to skim the text and extract structured information or "facts". In this context facts are interesting pieces of information like dates, locations, people, relationships and events. Although the system may be deployed against semi-structured text, and text embedded in other data streams, its focus is on processing human readable text in the form of short documents, such as news stories.

The system can be applied to an information extraction need as a stand-alone application[1] or embedded into a larger system through its published Application Programming Interface (API). Either approach can be automated or used interactively. In its interactive mode it provides a useful mechanism for perusal of documents by displaying the document with pertinent facts highlighted thus quickly drawing a user's attention to relevant areas. The system could also be deployed to automatically collect information in a format suitable for further analysis; for example, by application of data mining tools, or as an input to an indications and warnings system.

A key feature of the DSTO Fact Extractor System is the ability to tailor its use for a particular information extraction need. An interactive development environment known as the DSTO Fact Extractor Workbench [1],[3] has been produced to assist the mechanics of this task but it remains somewhat of an art to craft specific fact extractors. The goal of this paper is to provide some guidance on this task.

This paper will introduce a range of information extraction tasks and discuss how the intended use of the information might affect the design process. The paper will then define a range of concepts central to the fact extractor system and introduce some basic fact extractor building blocks, and common techniques that often get used when building real fact extractors. The core of the paper proposes a fact extractors design and development process and then works through a number of case studies. The paper concludes with a discussion on evaluation and future considerations.

---

[1] The principal user application is known as FormFiller[6]

2. Types of Information Extraction

Information extraction is a very broad field with examples as simple as finding geographic place names through to building up a picture of a terrorist organisation. It may deal with extracting information from just one document or attempt to exploit a large document collection. The DSTO system is aimed at extracting information from individual documents[2]. Other tools[3] may be used to fuse information from across a collection of documents. The DSTO system can be used against relatively simple "who, what, when or where" type information extraction requirements, commonly referred to as named entity extraction. The system can be also used to recover pronoun and other co-reference information in the text and to extract more complex concepts such as the relationships between named entities.

2.1 Named Entity

The simplest type of information extraction is to look for *names* of objects in the text. These might be the names of people, places, and organisations but also includes more abstract *names* such as currency amounts, dates and times. This type of information extraction is commonly referred to as Named Entity extraction and is of particular interest to Defence as it encompasses discovery of who, what, when, where information.

A variety of techniques can be used to discover named entities. Finding names of people or places can be achieved by matching against a list of desired names. Of course this technique will only work if you know beforehand the names that you are looking for. To find arbitrary names in a text can involve part-of-speech analysis or the use of other clues such as titles and capitalisation.

Objects like dates and currency amounts are not amenable to the list approach; these kinds of requirements are best addressed with a rule-based approach. For example a rule might say, "look for a currency symbol followed by a sequence of digits".

Named entity extraction generally only has to consider the text on a sentence-by-sentence basis as the named entity is nearly always represented inside a single sentence.

2.2 Co-reference

Information extraction also needs to concern itself with understanding co-references, for example pronouns like "he" / "she" and name abbreviations. Ideally the extraction process would link these co-references back to the original object; for example, it would

---

[2] Strictly the system can also access meta-data about the document, this could be used to combine information across documents but this is not the intent of the meta-data support.

[3] For example the Data Mining & Visualisation Toolkit, formerly known as the Normalcy Analysis Toolkit [13].

discover that "Tom" is a co-reference to the previously introduced person "Mr. Tom Smith".

Co-references may appear within a sentence or occur over a group of sentences. The DSTO system can deal with both cases.

2.3 Concepts

Information extraction doesn't stop at finding the named entities. Often of greater interest are relationships between the entities, which we refer to as "concept extraction". Concept extraction is concerned with discovery of events or relationships of interest. Examples include the location, date and type of events recorded; individuals involved in the events; and relationships between individuals.

Concept extraction is considerably less precise and more challenging than named entity extraction. As it normally relies on using information discovered by a named entity extraction, its effectiveness will be highly dependent on the effectiveness of the underlying named entity extraction. Unlike named entities that tend to exist in a single sentence, concepts are frequently embodied in a number of sentences. This makes defining the concept extraction process inherently more difficult and highly dependent on good co-reference resolution.

The DSTO system includes specific support for constructing an information system that can extract concepts between previously discovered named entities. Subordinate fact extractors that can discover desired named entities are often used to assist a composite fact extractor to discover an interesting concept. Particular types of rules are used to recover concepts. Subordinate and composite fact extractors are described in section 4.3 and fact extractor rules are described in sections 4.2.2 and 4.4.3.

3. Application of Fact Extractors and Accuracy Considerations

Fact Extractors can play many roles in assisting users to meet their information needs. These range from relatively lightweight tasks, such as highlighting an interesting fact to the reader, through to sophisticated automatic population of a structured database.

The cost of developing a particular fact extractor needs to be balanced against the value achieved by meeting the information need of the users. Experience has shown that striving for high accuracy entails a high cost in developer effort. Section 3.1 defines accuracy and Sections 3.2 to 3.6 describe some information extraction tasks and their accuracy needs.

3.1 Accuracy and Information Extraction

It is difficult, if not impossible, to construct Fact Extractors that discover all facts accurately. Accuracy considerations fall into several categories:

- **Under generation (false negative)**: Did the fact get missed altogether?

- **Over generation (false positive)**: Was a "non-fact" erroneously discovered? Did something get selected which should not have contributed to a fact?

- **Internal accuracy (precision)** of the fact: Facts are composed of a number of attributes, for example hours, minutes and seconds for a "time" fact. It is possible to only discover some of them correctly. These can be examples of false negative and false positive errors, but for a part of a fact rather than the whole fact.

- **Co-references**[4]: Were co-references to the fact under or over generated?

- **Implied information**[5]: Was implied information correctly found?

Over or under generation of facts is frequently a trade-off; often the Fact Extractor developer will need to make a conscious choice between the two cases, as the goal of never incurring a false positive or a false negative may be impossible over a wide range of source documents unless the fact extractor developer can examine every document in the corpus. Internal accuracy of a fact is more often related to the amount of effort the developer is prepared to expend on the development process. It must be kept in mind that achieving a very high accuracy with any information extraction tool may be a very time consuming task. The developer needs to know when to stop and accept the outcome, particularly as many tasks are tolerant of certain types of errors. For example attempting to capture misspellings to avoid false negatives in real documents may result in a higher false positive rate by matching other words.

3.2 Assisted Reading

Frequently there is a requirement for users to peruse large volumes of documents seeking out pertinent facts. Fact Extractors can assist by highlighting these facts as the documents are presented one at a time to the user, quickly drawing the user's attention to salient portions of each document. In this scenario a user is unlikely to be offended by a moderate degree of over generation provided that there is little under generation. The internal accuracy of the attributes of the fact is irrelevant, as users will not actually look at the fact, rather they will read the sentence/paragraph that contains the fact. Pronoun discovery (an example of co-reference) is likely to be important, but again a degree of over generation will be tolerated.

---

[4] A Co-reference is a reference to a previous concept. For example the pronoun *he* would normally refer to a previously introduced male.
[5] Often information is implied rather than explicitly stated. For example the year part of a date is often left out when it is obvious.

The FormFiller[6] application distributed with the Fact Extractor System can be used in an assisted reading mode. A novel feature of the FormFiller application is the ability to apply several fact extractors to the document at the same time. This saves the fact extractor developer from needing to develop an elaborate all encompassing fact extractor leaving it to the reader to turn on an appropriate collection of simpler fact extractors. In this way the often-complex task of determining the relationships between facts is left for the reader to do as they read the highlighted sentences rather than requiring this issue to be solved by the fact extractor developer.

3.3 Advanced Information Retrieval

Full text retrieval tools such as the popular Internet search tool Google™[7] often return many thousands of documents against a typical search. The Fact Extractor technology could be used to reorder[6] the result set in an effort to increase the probability that more important documents are closer to the top. A user might perform a Google™ search with a number of keywords. A fact extractor could then be used to look for a particular relationship between the keywords and thus reorder the result set based on the facts found. In this example fact extractors are being applied to an already imperfect information source (the result of a search) and as the fact extractor is not discarding information it is likely that the user will be very tolerant of fact extractor errors. The internal accuracy of the fact is largely irrelevant and it is unlikely that co-reference discovery will be required. Even quite severe over-generation will not be a problem; as long as the important documents are raised in priority, the reordered result set will be better than the original result set. Often a moderate degree of under generation will also be acceptable as long as an example of the fact is found; there is no need to find all examples in the same text. Frequently the user only wants to find a reference to a subject, rather than all references. In this case the fact extractor can significantly under-generate results but provided a fact somewhere from the document collection is found it will still be of assistance to the user.

3.4 User Assisted Database Population

The business processes of an organisation might require users to populate a database with facts from a collection of documents. The FormFiller[6] application can perform user-supervised population of a database with facts from a target document set. Each document in the set is presented to the user with all discovered facts highlighted. Attribute values are computed on-the-fly as the user clicks on particular highlighted facts. The application supports the user manually modifying the attribute values computed by the fact extractor. When the user is satisfied with the extracted information they may choose to send the results to the database.

The accuracy requirements for user assisted database population are similar to the assisted reading task described in section 3.2. In particular the correct discovery of co-

---

[6] At this point in time the fact extractor team has not produced an example application for this task.

reference and implied information is unlikely to be important, as the user would generally add this information. The principal difference is that the user will now pay attention to the fact attributes, and will therefore expect greater internal accuracy. While some correction might be acceptable, the user will not want to have to correct every attribute of every fact.

As for assisted reading, the FormFiller application is able to apply several fact extractors to the document at the same time. Again this saves the fact extractor developer from needing to develop an elaborate all-encompassing fact extractor leaving the reader to pick out the appropriate collection of related simple facts. The FormFiller application allows facts from several different fact extractors to be combined into one database record.

3.5 Automatic Database Population

If the Fact Extractor is robust and sufficiently accurate a user may choose to automatically populate a database with all discovered facts from the target document set. The FormFiller[6] application can be used for this task. Alternatively the Fact Extractor core library[8] may be used to create a custom application.

In this example fact extractors are used in an unsupervised manner so the accuracy constraints become tighter. The likely outcome is that only relatively simple fact extractors with modest information extraction expectations would achieve the required accuracy. While it is possible to use the FormFiller application in an automatic mode with more than one fact extractor selected, more satisfactory results will be achieved with a single, purpose-designed fact extractor.

3.6 High Volume Data Feeds

Defence agencies often have access to many dynamic information feeds. They range from the largely uncontrolled Internet news groups through to paid news subscription services. Many of these information feeds do not provide adequate (if any) metadata for each information item or article. The challenge is the volume, much of the material is not of immediate interest and it is not practical to have analysts routinely monitor many of these sources.

Sometimes there will be articles which mention people, places, or events of potential interest. An application could be developed that uses Fact Extractors to find potentially interesting facts. These facts can be considered to be metadata about the article. This metadata could be used to populate a database, including references back to the news articles from which they are derived. Once the data is in a structured form, further analysis, such as data mining, can be used to discover information which may not have been available from reading the individual sources. The application could also generate alerts if it finds facts about particular interesting concepts.

Given the volume of information, fact extractors developed for this task will need to trade off accuracy for performance. Over generation (creation of non-facts) should be avoided as is likely to mislead subsequent data mining applications.

4. Fact Extractor Concepts

Before we can explore the process of building fact extractors we need to define some concepts which are core to the fact extractor system. In this section we will introduce *facts*, *fact extractors* and the components of a fact extractor: the *rules*, *guards*, *patterns* and *actions*. We will also provide an overview of the fact extractor processing model.

4.1 Facts

In the context of the DSTO Fact Extractor System a fact is information about a particular entity, relationship or event of interest. A fact is represented by:

- The name of the type of fact.

- A list of attributes that define the fact and their corresponding values (if discovered).

- The text location that contain the evidence that created the Fact.

- Sequences of other text locations, which refer to or extend the evidence about the fact.

4.2 Fact Extractors

Fact extractors are used to discover facts. A **Fact Extractor** is implemented through a generic processing engine and a specification that defines how it skims text, produces facts and assigns values to attributes.

A Fact Extractor includes:

- A list of field names representing the attributes of the fact.

- A list of rules that describe how a fact is found and the attributes calculated.

To simplify the construction of complex information extraction systems the DSTO system supports building Fact Extractors in a hierarchical manner allowing higher-level Fact Extractors to invoke subordinate ones.

4.2.1 Fact Attributes

A Fact Extractor Specification includes a list of field names that represent the attributes that the system will attempt to populate from the text. For example a date fact would typically have day, month and year fields.

4.2.2 Fact Extractor Rules

The Fact Extractor specification includes a set of pattern and action rules. The *patterns*, expressed in a modified Regular Expression Language[4],[8] are used to detect the existence of a fact. The *actions* calculate values for attributes of the identified fact.

Each rule has:

- A type – *create*, *co-reference* or *expand* – described below.

- A pattern represented by a modified regular expression.

- A guard expression. A guard expression may provide extra constraints to the pattern such as range-checking a number. It can also reference previously found facts. The supported functions are described in Appendix B.

- A list of actions. An action is represented by a function which computes a value. One action is specified for every field for which a value is to be determined by that rule. The supported functions are described in Appendix B.

The rules are grouped into three types:

Create Rules:	Describe the patterns that identify the fact in the text. Each create rule includes a specification of the action to be undertaken if the pattern matches. A new fact is identified every time a create rule matches the text.
Co-reference Rules:	Specify connections back to the most recently discovered fact of this type that satisfies the guard expression. For example pronouns like "she" and "he", people later referred to only by their first name, or role-based references such as "the president".
Expand Rules:	Update or add to the attributes of an existing fact. For example a section of text like "…has blue eyes…" would not be matched by a *person* create rule as this is insufficient evidence for a person, but this text adds information about eye colour. Expand rule patterns must contain the special pattern `<self>` which identifies where a suitable create or co-reference rule has matched in the text.

Each rule type is subject to a particular processing model described in Section 4.4.3

4.3 Categories of Fact Extractors

As described earlier a key feature of Fact Extractors is that they may use other Fact Extractors. There is some design and testing differences between fact extractors that are intended to be used by other fact extractors and those that are not. This section introduces some categories of fact extractors; these terms will be used later in the document.

Primary   A Fact Extractor that is not intended to be invoked by any other Fact Extractor is known as primary. Its field names will be aligned with the information need of the task.

Composite   A fact extractor that uses another fact extractor is known as a composite Fact Extractor.

Subordinate   A fact extractor that is invoked by another fact extractor is known as a subordinate fact extractor. Fact extractors may be invoked by more[7] than one composite. Subordinate fact extractors will need a broader testing regime.

Simple   A fact extractor that does not invoke any other fact extractors is known as a simple fact extractor.

A fact extractor can exist in more than one category. A single fact extractor that independently satisfies a particular information need is both primary and simple. Many simple fact extractors would be routinely used as subordinates.

4.4 The Fact Extractor Processing Model

Some knowledge of how Fact Extractors work [8] is required to be able to develop effective fact extractors. Fact extractors are applied to a document a sentence at a time. If a particular fact extractor invokes other (subordinate) fact extractors then these subordinates are processed first. Any subordinate facts found are retained in the fact cache to be described in section 4.4.4.

4.4.1 Finding the Sentences

Fact Extractors process one sentence at a time. The input document is broken up into sentences by a piece of software known as the sentence-manager. The sentence-manager uses clues such as capital letters and full stops to determine the sentence boundaries. It also uses the block structure of documents (such as paragraph elements in HTML) as indications of sentences. At times the behaviour of the sentence-manager will not conform to the user's expectations; it is proposed that a future version of the fact extractor system will allow the fact extractor developer to fine-tune the sentence-manager's behaviour.

4.4.2 Reformatting Text

The Fact Extractor system reformats plain text to make it easier to write patterns against the intent of the sentence, without needing to consider that the layout happened to insert a line break in the flow of text. It also removes any extraneous white space for the same reason. It attempts to intuit paragraph breaks from the layout, and

---

[7] The Fact Extractor Workbench[12] includes a tool that lists all of the composite fact extractors that use a particular subordinate fact extractor (it's subordinates are also listed).

leave them intact. This is usually correct, but may not be right for some partially-formatted inputs.

4.4.3 Processing a Sentence

For each sentence, the processing engine performs the following steps in order.

Subordinate Fact Extractors	If any of the primary fact extractor's rules refer to subordinate fact extractor(s), then all the subordinate fact extractors are run completely (according to the process in this section) before the primary fact extractor. This places their results in the fact cache so they are available for the primary fact extractor.
Create Rules	Create rules create a new fact. All of the create rules are evaluated in order. To evaluate a rule, its pattern is used to find any matches on the text in the sentence. For each match, the guard expression is evaluated, and if it is TRUE (or there is no guard expression), the rule's actions are performed to assign values to the fields in a newly created fact. If more than one create rule for a fact extractor matches the same part of the text, only one fact is created. If the matched range of one completely contains the range of the other, then the rule with the longest match is the one that creates the fact. If the ranges are exactly equal or overlap, then the last create rule is the one used, on the assumption that the rules are listed from the simplest to the most complex. In practice, it is very rare to have different ways of finding the same fact that rely on overlapping parts of the text.
Co-reference Rules	Co-reference rules identify another reference to an already-discovered fact. Co-reference rules are evaluated after create rules, updating an existing fact, rather than creating a new one. Co-reference rules usually have a guard expression to restrict which fact might be referred to, but often have no direct actions (they automatically update the ranges of text referred to). They search back through the fact cache to find the most recent fact of the right type that allows the guard expression to evaluate to TRUE. All co-reference rules are evaluated in order.
Expand Rules	Expand rules add information to a previously-found fact. They are evaluated after co-reference rules so that they can add attributes to a fact regardless of whether it is an original create rule match or a later co-reference that is being used to anchor the rule in the text. Expand rules usually have actions to update one or more fields of the fact with additional information. Guard expressions may be used in expand rules in the same way as in create rules.

4.4.4 The Fact Cache

For simple named entity extraction, such as a place name, it may only be necessary to examine the sentence under consideration. However for more complex information extraction tasks the concepts introduced in previous sentences have a bearing on the information in the sentence under consideration. One example is the use of a pronoun such as *he*, which in a well-constructed text would refer to a previously introduced male. Another example is a partial date like *27th of February*. In this case the year is missing, but this should be evident from the year of a previous date reference or from document metadata such as publication date. In order to support the development of fact extractors that cope with such examples all of the facts discovered so far are kept in a fact cache. Facts in the fact cache can be accessed[8] during the evaluation of a guard expression or during the processing of the action part of a rule.

All of the facts discovered are retained in the cache, including those discovered by subordinate fact extractors. When a document has been processed only the top-level facts are reported and then the cache is cleared. At any point in time therefore the cache only holds facts and subordinate facts pertinent to the document currently being processed.

5. Fact Extractor Design Patterns

The process of constructing useful fact extractors may appear quite complex but usually involves combining some fairly basic concepts. Examples of these concepts are finding a word from among a list of words, or looking for a pattern that represents a date or currency amount. This section introduces those basic concepts or building blocks (known as *design patterns* in the software industry); the next section shows how they are applied to a range of real information extraction problems.

An overview of the pattern language is provided in Appendix A and a summary of the action language is provided in Appendix B

5.1 A Simple List with Just a Few Items

A fairly common information need is to find instances of certain words in a document and return the word as a fact. For example we may need to find occurrences of January, February, … and record them as a month. A fact extractor could be constructed with a single field Month, and the following create rules:

```
Pattern: January      Action: Month = "January"
Pattern: February     Action: Month = "February"
Pattern: March        Action: Month = "March"
…
```

---

[8] The action language function CACHE provides this support, and the FXBench debugging environment allows a developer to view all of the cache contents during development.

An alternative to this approach is to combine the 12 rules into one and use the actual text matched as the result.

```
Pattern: January|February|…|November|December
Action: Month = MATCH("0")
```

Concepts introduced:

Plain text as the pattern and text strings in the action.
"|" as the regular expression "or" function.
MATCH("0")[9] returns the text matched by the rule.

5.2 A Simple List with Result Substitution

At times you may need to match items in a list but return something different from what was matched. For example you may wish to find Jan or January and return "1". This can be achieved with create rules like:

```
Pattern: Jan|January            Action: MonthNum = "1"
Pattern: Feb|February           Action: MonthNum = "2"
…
```

The above patterns could be rewritten in a more condensed form:

```
Pattern: Jan(uary)?             Action: MonthNum = "1"
Pattern: Feb(ruary)?            Action: MonthNum = "2"
…
```

Concepts introduced:

Assignments in action sections can be different to the text matched.
Parentheses for grouping.
"?" for matching 0 or 1 occurrences of the previous group or character.

5.3 A List with Many Items

If the concept is extremely simple, and all that is required is to find matches for a list of words or phrases, there is a special type of fact extractor called a *list fact extractor*. These have a much simpler file format (simply a list of phrases in a text file, one on each line). This file must be named to end with ".fx" and may be created with any text editor. If the phrases are all lower case, they will match any case in the text; if there are upper case letters in the list, then matching is case-sensitive. The Fact Extractor Workbench provides a simple editor for these lists. They may be processed using different internal algorithms, which are intended to be optimised for these lists. When list fact extractors are used, they extract facts with a single field containing the phrase that matched.

---

[9] In general MATCH("n") returns the nth parenthesised group in the rule, MATCH("0") is a special case that returns the whole text matched by the pattern.

Concepts Introduced:

List fact extractors.

5.4 A Simple Pattern

The list examples above introduced two of the operators that can be used in a pattern: "|" for alternation and "?" for zero or one occurrence of the previous group. This section introduces the power of relatively simple patterns. Suppose that we desire to locate formal references to people in a document. We might do this based on looking for a title (Dr. Mr. Mrs. Miss. Ms.) followed by a word starting with a capital letter. If we had two fields called `Title` and `Surname` we could use the following rule:

```
Pattern: (Dr|Mr|Miss|Ms|Mrs)\.? ([A-Z][a-z]+)
Action: Title = MATCH("1")
Action: Surname = MATCH("2")
```

Fact Extractor regular expressions can be more complex. The reader is referred to the online help provided in the Fact Extractor Workbench for a discussion on Regular Expressions, and to [4],[5] for a detailed coverage of Regular Expressions.

Concepts introduced:

+ for matching one or more times.
– for a character range.
[ ] for a disjunction (exactly one of the characters to match).
Using MATCH("n") to extract the text that matched the nth parenthesised expression.
\ to quote a special character such as ".", "[", "(", "?".

5.5 Multiple Create Rules

The above examples will find Mr. Smith but will not find names like Mr. Bill Smith or Ms Mary O'Shea. To also find these names, we could change the fields used to Title, Firstname and Surname, and use the following two rules:

Rule 1: (Finds a name with a title and surname.)

```
Pattern: (Dr|Mr|Miss|Ms|Mrs)\.? ((Mac|Mc|O')?[A-Z][a-z]*)
Action: Title = MATCH("1")
Action: Surname = MATCH("2")
```

Rule 2: (Finds a name with a title, first name and surname.)

```
Pattern:
   (Dr|Mr|Miss|Ms|Mrs)\.? ([A-Z][a-z]*) ((Mac|Mc|O')?[A-Z][a-z]*)
Action: Title = MATCH("1")
Action Firstname = MATCH("2")
Action: Surname = MATCH("3")
```

When more than one create rule is defined for a fact extractor, the rules are processed in order (see Section 4.4.2). If the text matched by different rules overlaps only one fact will be created as follows:

- When the text matched by a rule completely contains that matched by any other rule(s), the rule with the longest match is the only one that will create a fact. For example consider the text string "Jack and Jill went up the hill". Consider the case where there are three rules – rule 1 matches "Jack and Jill"; rule 2 matches "Jack"; and rule 3 matches "Jill". Note that the text matched by rule 1 completely contains that matched by rules 2 and 3. Therefore the fact created by rule 1 ("Jack and Jill") will be discovered.

- When the text matched by a rule overlaps (or exactly matches) text matched by any other rule(s) then the last matched rule is the only one that creates a fact. For example, with the string given above, consider two rules – rule 1 matches "Jack and"; and rule 2 matches "and Jill". Note that in this example the text matched by the two rules overlap. Therefore, as rule 2 came after rule 1 the fact created by rule 2 ("and Jill") will be discovered.

Concepts introduced:

Fact Extractors can have more than one create rule.

5.6 Complex Actions

In the examples introduced so far the action part of the rule has simply set a field to a fixed value or used the actual matched text as the value. Sometimes you may want to put together information to derive the complete value of a fact. For example, to extract the full name of a person you could put together their title, first name and surname using the `CONCAT` action function. Using the example discussed in the previous section for finding person names, an additional action is used to compute a value for the `Fullname` field:

Rule 2:

```
Pattern:
   (Dr|Mr|Miss|Ms|Mrs)\.? ([A-Z][a-z]*) ((Mac|Mc|O')?[A-Z][a-z]*)
Action: Title = MATCH("1")
Action: Firstname = MATCH("2")
Action: Surname = MATCH("3")
Action: Fullname = CONCAT(MATCH("1")," ",MATCH("2")," ",MATCH("3"))
```

A variety of other functions are listed in Appendix B, they may be combined together.

Concepts introduced:

The expressive power of the action language.
CONCAT(Argument1, Argument2, …) which returns the concatenation of its
       arguments.

5.7 Implied Information

Frequently in text some information is left out but is implied from surrounding text. A complete date[10] is generally expressed in terms of a day, month and year. Sometimes dates are only partially stated in the text, for example "4 Feb". When this occurs a reader would typically reference the previous more fully specified date to resolve the year for "4 Feb". The action function CACHE looks in the fact cache (refer to Section 4.4.4). It can be used to complete implied information such as the year in a date, if a complete date has been specified in a previous sentence of the current document. A Date fact extractor that extracts dates of the format "12 Feb 02" could have the following pattern and action rules:

Rule 1:

```
Pattern: ([1-3]?[0-9]) (Jan|Feb|…|Dec) ([0-9]{2})
Action: day = MATCH("1")
Action: mon = MATCH("2")
Action: year = MATCH("3")
```

We can extend this fact extractor to extract partially stated dates of the format "3 Feb" and compute the year based on the most recent previously found Date fact in the document. To do this we can add another rule[11] to the fact extractor that uses the CACHE function as follows:

Rule 2:

```
Pattern: ([1-3]?[0-9]) (Jan|Feb|…|Dec)
Action: day = MATCH("1")
Action: mon = MATCH("2")
Action: year = CACHE("Date","year")
```

Concepts introduced:

{n} to match exactly n occurrences
CACHE(<fact extractor name>, <fieldname> [, < default value>]) which searches the
     cache of recent facts for the specified fact extractor name and returns the value
     of fieldname (or an optionally specified default value if not found).

5.8 External Action Functions

From time to time there may be a requirement in the action part of a rule that cannot be met by the supplied action functions. A Java programmer can add new actions[12] to the

---

[10] Dates are a difficult concept and are discussed in more detail in Section 7.2

[11] Rule 2 will only create a fact if Rule 1 has not as it matches a shorter text. See Sections 4.4.2 and 5.5 for more details.

[12] The Online help on FX Expression Language in the Fact Extractor Workbench [12] provides information on how external functions can be invoked as Action functions. Reference [8] also discusses External Action functions.

collection of allowable actions. These user-defined extensions can then be used as a part of the action language to compute field values.

Date references that require calculations, such as "yesterday", are an example of an extraction problem that may need an external action function.

Consider the following excerpt from a news story taken from [9]:

GUATEMALA CITY, 4 FEB 90 (ACAN-EFE) -- [TEXT] THE GUATEMALA ARMY …ATTACKED THE FARM 2 DAYS AGO.
…

Notice that there is a date present in the first sentence of the news story. Our information extraction task may require us to capture the date of any attacks. The text "2 DAYS AGO" is clearly a reference to a date, but resolution and calculation[13] is required to discover the actual date. An external Java function, for example AddDaysToISODate(), could be developed that would allow arithmetic on dates.

If we had a basic ISODate[14] fact extractor with a single field date then the addition of the following rule would support matching text like "2 DAYS AGO".

Rule:…

```
Pattern: ([1-9][0-9]?) DAYS AGO
Action: date = AddDaysToISODate (CACHE("ISODate","date"),
            MULTIPLY("-1",MATCH("1")))
```

When invoking the AddDaysToISODate() function, we have used the CACHE Action function to obtain the most recent previously discovered date. Since we are trying to subtract a given number of days from a known date, we use the MULTIPLY Action function to convert the integer parameter to a negative number.

Concepts introduced:

The ability to extend the action language with external user functions.
MULTIPLY(argument1,argument2,..) which returns the product of its arguments.

5.9 Extra Constraints

At times the pattern language used for the rules may not be expressive enough to accurately constrain the facts produced. For example we may wish to find Internet Protocol (IP) numbers. IP numbers are four numbers each in the range 0-255 with a dot between each number. It is easy to write a pattern that finds a number in the range 0-299; it is rather harder to write one that finds a number in the range 0-255[15]. The author

---

[13] The calculation is non trivial and involves (at least) consideration of leap years.

[14] We have kept this example simple by using ISODate which uses a date representation[10] defined by the International Standards Organisation. In this example a single field of the form yyyy-mm-dd represents a date.

[15] In section 7 we will discuss the need to examine sample documents before developing a fact extractor. If we examine a range of sample documents it is likely that we would never see an example of something that has the same pattern as an IP number but that does have some numbers in the range 256-299. So this level of guard check is most likely pedantic and unnecessary.

of an IP number Fact Extractor may find it easier to look for groups of numbers in the range 0-299 and then use a guard expression to further constrain the selection to 0-255.

```
Pattern: ([12]?[0-9]?[0-9])\.([12]?[0-9]?[0-9])\.([12]?[0-9]?[0-
         9])\.([12]?[0-9]?[0-9])

Guard:   AND ( LESSTHAN(MATCH("1"),256),
               LESSTHAN(MATCH("2"),256),
               LESSTHAN(MATCH("3"),256),
               LESSTHAN(MATCH("4"),256) )

Action:  IP = CONCAT(MATCH("1"),".",MATCH("2"),".",MATCH("3"),
                 ".",MATCH ("4"))
```

Alternatively

```
Action:  IP = MATCH ("0")
```

A second example is that we may wish to tighten the specification of our date fact extractor to create facts for text patterns that are valid dates, including correctly checking the number of days in the month. A guard expression can be used to only allow the fact to be created if it will be a truly valid date.

Of course to do this properly we should also check for leap years. Leap year rules are quite complex and beyond the scope of what the fact extractor action language was designed for. They are however a good candidate for an external action function. If we had a Java programmer develop a date_valid()[16] function we could then write a fact extractor like:

```
Pattern:…
Guard: date_valid(MATCH("1"),MATCH("2"),MATCH("3"))[17]
Action:  day = MATCH("1")
Action:  month = MATCH("2")
Action:  year = MATCH("3")
```

Concepts introduced:

Guard expressions in create and expand rules.

5.10 Subdividing Complex Tasks

A key design principle of the DSTO Fact Extractor System is to support decomposition of complex tasks into simpler parts. Composite fact extractors are able to use the results of other fact extractors. This was described as part of the processing model discussion in Section 4.4.2.

---

[16] In this example date_valid() expects three strings representing the day, month and year, returning true or false as appropriate.

[17] Note that the values of the fields day, month and year have not been calculated at the time of performing the guard check so we must use the MATCH function to obtain the parameters to the date_valid() function.

Subordinate fact extractors are invoked by including the name enclosed in angle brackets ("<…>") in the pattern of a rule. Their values may be accessed in actions and guard expressions using the FIELD(…) function. For example a simple Date[18] fact extractor might rely on a Month fact extractor (such as the one introduced in Section 5.2).

```
Pattern: ([0-3][0-9]) (<Month>) (([12][0-9])?[0-9][0-9])
Action: day = MATCH("1")
Action: month = FIELD("2", "MonthNum")
Action: year = MATCH("3")
Action: DateString = CONCAT(VALUE("year"), "-", VALUE("month"), "-
        ", VALUE("day"))
```

Subgroups in the regular expression pattern are numbered in order from left to right, counting the **opening** parenthesis ("("). In this example pattern, the century number can be accessed as MATCH("4"). If there was no century in the text, the match will be an empty string.

The current implementation does not support references to subordinate fact extractors inside option constructs such as alternative ('|'), optional ('?', '*', '+') or look-ahead and look-behind. For example <Month>? is not supported. Please refer to the Fact Extractor Workbench online help [12] for more details.

Concepts introduced:

Subordinate fact extractors represented as <subFXname>.
FIELD() function to recover one field from a subordinate fact
Nested subgroups.
VALUE() function to access another field from the current fact.

5.11 Linking Back to Previous References of a Fact (Co-reference)

Assisted reading only requires highlighting those parts of the text which match some patterns. To accurately populate a database, a deeper analysis is required. This includes being able to recognise whether a new reference is introducing a new fact, or an additional reference to an already discovered fact. Co-reference rules allow the fact extractor developer to specify that some fragments of text are just further references to a fact discovered earlier. This is frequently necessary to make sure that additional information collected by expand rules is assigned to the right fact.

As the pattern in a co-reference rule is necessarily vague, co-reference rules usually have a guard expression to ensure that the right fact is found in the fact cache.

For example, the Person fact extractor described in section 5.6 had three fields (`title`, `firstname` and `surname`). The person might be referred to later by just their first name. A co-reference rule to capture this would have a pattern and guard expression such as

---

[18] This is deliberately a simple example that doesn't consider all cases such as guessing the century for a 2-digit year. Refer to Section 7.2 for a more complete Date example.

```
Pattern: [A-Z][a-z]+
Guard: EQUAL(VALUE("firstname"), MATCH("0"))
```

The pattern is quite vague, and matches any single word starting with a capital letter. The guard expression is used to limit the rule to only apply if the matched word has already been identified as a person's first name. The rule will search back through previously found Person facts and evaluate the guard expression for each Person fact until it finds one that allows the guard expression to be true, or runs out of previous facts. When a person fact that satisfies the guard expression is found, then this piece of text is added as another reference to that person, and any actions in the co-reference rule are performed on that fact. Co-reference rules can also be used for pronouns.

Concepts introduced:

Co-reference rules.
Guard Expressions in co-reference rules search back through previous facts.

5.12 Expand Rules

Often the attributes of non-trivial concepts are spread over several sentences. Generally it is possible to decide what is the key aspect and a fact extractor create rule can be developed to extract a fact based on it. However create rules can only be used to find a fact within one sentence. Fact Extractor Expand rules are used to recover extra attribute information gleaned from subsequent sentences. They may also be used to recover extra optional information from within the same sentence.

The expand rule pattern must describe the connection back to the previously found fact that it is expanding. The special reference of `<self>`[19] is used to refer to a previously found fact. The connection to the previous fact could come from either a create rule or a co-reference rule.

An example is a hospital visit fact extractor looking for information from the text

> "Mr Smith was taken to hospital with a high temperature. The patient was also complaining of a severe headache. He also had a rash."

If the information to be extracted is to be used to complete a hospital visit record, clearly the first sentence has a hospital visit fact and a create rule should be used to recover this information. The second and third sentences have extra information about the admission but neither of them should create a hospital visit fact. Expand rules, in conjunction with co-reference rules, are used to recover this supplemental information.

Returning to the example, the `HospitalVisit` fact extractor has two fields: `symptoms` and `patientName`. Two subordinate fact extractors are also used. They are `person` and `symptom`; their description is omitted for brevity. The `HospitalVisit` fact extractor has a create rule that looks for evidence of a hospital visit. Clearly a real implementation would have a number of create rules.

---

[19] <self> is the same notation as a subordinate fact extractor but the behaviour is quite different.

Create rule 1:
```
Pattern: (<person>) was taken to hospital
Action: patientName = FIELD("1", "FullName")
```

In this example the `HospitalVisit` fact extractor has two co-reference rules. The first looks for references characterised by the term "the patient". The second co-reference rule uses the subordinate person fact extractor to find references to a person and then uses a guard expression to ensure that the person name matches the patient's name. This rule does not specifically identify whether the patient is identified by name or a pronoun, that task is left to the subordinate person fact extractor.

Co-reference Rule 1:
```
Pattern: [Tt]he patient
```

Co-reference Rule 2:
```
Pattern: (<person>)
Guard Expression: EQUAL(FIELD("1", "FullName"),
   VALUE("patientName"))
```

The two expand rules collect extra information about the hospital visit, either from the sentence that invoked the create rule or from subsequent sentences.

Expand Rule 1:
```
Pattern: <self> with (<symptom>)
Action: symptoms = CONCAT(VALUE("Symptoms"), ",  ", MATCH("1"))
```

ExpandRule 2:
```
Pattern: <self>( (is|was|has))?( (also|sometimes|often))?
   (complain(ing|ed) of|reported|had) (<symptom>)
Action: symptoms = CONCAT(VALUE("symptoms"), ",  ", MATCH("7"))
```

Concepts introduced:

Expand rules.

5.13 Reducing the Range of a Match

It is sometimes useful to be able to use the pattern to match more text than is required by the final fact. For example a person fact extractor that uses a rule based on capitalisation may include in the rule a set of verbs that a person is likely to do; however this would result in a fact that is "too long" in terms of the amount of text it matches. The text extent of the match is important if the range is being used for marking up the document for display, or saving the original text. It also matters if later expand rules extend a fact, as they need to find the fact at the right place in the text.

The fact extractor system has a way to reduce the range of the matched text provided pattern groups "(…)" are used in the expression. The "special field" `matchrange` may be assigned a pattern group number which reduces the recorded text extent to only that pattern group.

```
Pattern: ([A-Z][a-z]+) (<verb>)
Actions: matchrange = "1"
         Name=MATCH("1")
```

Concepts introduced:

matchrange.

5.14 Execution Speed Considerations

This section provides an overview of the performance issues that may need to be considered when using fact extractor design patterns; a separate report[14] provides a more detailed analysis of the execution speed profile of the DSTO Fact Extractor system. The following points should be taken into consideration.

- In general, performance will only be an issue for fact extractors run in an automatic mode. When developing fact extractors for an assisted reading or user assisted data base population task, performance is very unlikely to be an issue and can be safely ignored.

- There is a fixed cost in processing a document combined with a variable cost based on the complexity of the fact extractor(s) being applied. There is also a small cost for each fact found, including the facts found by subordinates.

- Each rule is evaluated for every sentence in the document, so a fact extractor with many rules will be slower than one with few rules.

- Create rules and Expand rules have a similar processing overhead. Co-reference rules can have a significant overhead for each pattern match as the guard expression is then evaluated for every existing fact. The pattern for a co-reference rule should therefore not be too loose if it can be avoided.

- In general, due to the way regular expressions are processed, longer patterns do not take significantly longer to execute. However, it is possible to construct specific examples that execute very poorly in certain circumstances[4].

- Subordinate fact extractors only have a minor overhead; just consider the number of rules in your subordinate and the likely number of facts they will generate. Use subordinate fact extractors if they simplify the extraction task. Be careful that subordinates don't end up generating many facts per sentence, particularly if the majority of these facts will not be used. Subordinate fact extractors can be tested in either FormFiller or the Fact Extractor Workbench to see how many facts they would create.

- In general patterns evaluate faster than guard expressions. It is inefficient to define patterns that frequently match and then use a guard expression to discard most matches. It is more efficient to put as much of the logic as possible into the patterns.

- When trying to match any word from a list of words consider using a List Fact Extractor (introduced in section 5.3). List Fact Extractors offer significant

performance gains over multiple trivial create rules or long patterns with many simple alternatives. If there are more than twenty items in your list it is sensible to employ a List Fact Extractor, perhaps as a subordinate fact extractor. If there are hundreds of items in your list it is highly desirable to use a List Fact Extractor for clarity and execution performance.

- Due to the ways that interactions with previously found facts are processed, documents that contain a very large number of facts (thousands) will suffer degrading performance. Create rules scan existing facts checking that a "larger" rule hasn't already matched the text in question. Co-reference rules scan existing facts backwards and re-evaluate the guard expression looking for a fact to co-reference. Expand rules scan existing facts backwards looking for the original fact to expand. This should not be an issue if fact extractors are used in their design domain, for example news stories, but may become a problem if applied to very large fact-rich documents, for example large log files.

6. Overview of the Fact Extractor Development Process

Information extraction systems can be time consuming to set up. One of the main reasons for this is that making machines understand unstructured text is hard – much harder than most people realise. The DSTO fact extractor system provides a set of integrated tools to assist a person to set up an information extraction system. Typically we would expect this to be the task of a knowledge engineer, someone who is familiar with the tool suite and the information extraction process. The knowledge engineer, in close collaboration with the subject matter specialist (end user), then develops the required set of fact extractors using the fact extractor workbench.

Although the Fact Extractor Workbench [12] does not prescribe any particular process to be followed when developing a fact extractor, the knowledge engineer will find their job is easier if they follow an organised process. Figure 1 provides an overview of the recommended steps involved in developing a fact extractor. The remainder of this section expands on these steps.
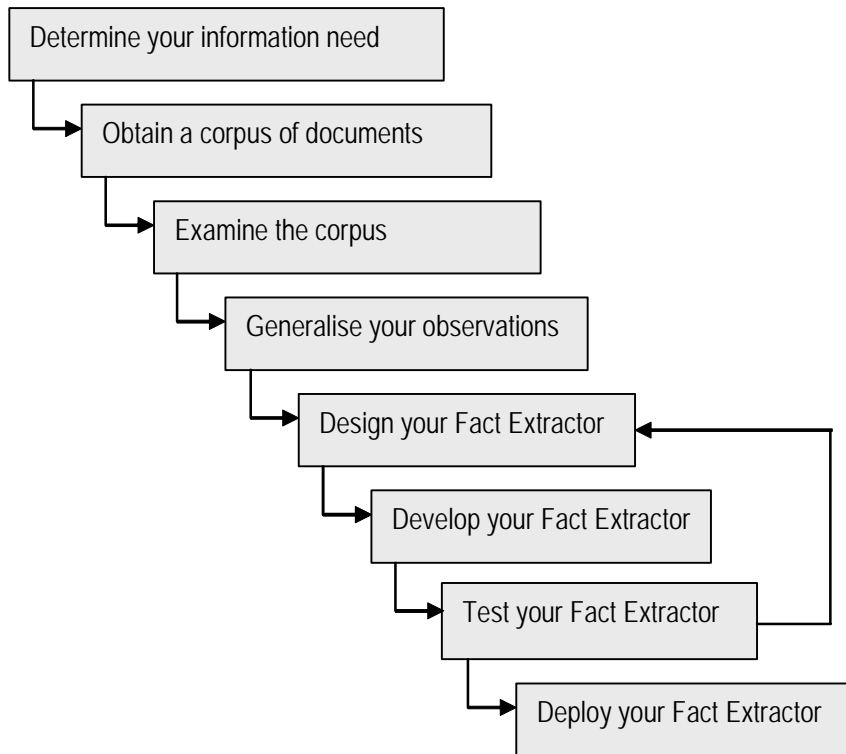
Figure 1: Overview of the Fact Extractor Development Process

6.1 Identify the Information Need

Before a fact extractor is developed the knowledge engineer must determine the information need. This will include consideration of:

- What information is to be extracted?

- What is a useful representation of the information?

- How will the information be used?

What is to be extracted needs to be agreed with the end user and is not considered further here. How the information is represented also needs to be decided in consultation with the end user, but it is worth taking into account possible future reuse of the fact extractor. For example, the end user may have no preference between representing a date as a string of letters or separating it out into day, month and year. However the day, month and year form may be beneficial to a subsequent user wishing to perform range calculations on the dates. How the information will be used will significantly impact on the accuracy requirements of the fact extractor. Fact extractor usage and accuracy implications were discussed in Section 2.

The goal of this first step is to identify a set of attributes that adequately cover the information need. These attributes are then turned into a set of field names and the

goal of the rest of the process is to develop the rules that populate field entries from the text. It is likely that the field names may be modified during the development process as certain representations of the information may turn out to be easier to work with[20].

6.2 Collect a Corpus of Representative Source Documents

Fact Extractors get information out of text, but are typically tailored to the text they will be used on. For this reason, before the knowledge engineer can begin to solve the information extraction problem, they need to get a significant and representative body of real text (a corpus) over which the information extraction activity is to occur.

Using a real corpus will allow characteristics of the text (perhaps a capitalisation convention) to be exploited. A real corpus should also allow the knowledge engineer not to bother developing rules that match theoretically possible patterns that turn out to not be in actual texts.

6.3 Examine the Corpus

Before commencing the development of the fact extractor it is recommended that the knowledge engineer examine a sufficient portion of the corpus to identify the forms in which the required information of interest may occur. It may be beneficial to print some interesting examples from the corpus, highlight the text that suggests a fact and write down the attributes that should be discovered. While examining the corpus the knowledge engineer should consider what sort of text would identify a new fact and what sort of text adds value to an existing fact. The results of this analysis should be discussed with the end user. It will often be found that there is not automatic agreement on what constitutes an interesting fact and some discussion may be required.

Since it may not be possible for the developer to examine every document in a large or growing corpus, it will be necessary to obtain a representative document subset. The "owner" of the corpus should be consulted on sampling strategies. Sampling considerations include the need for longitudinal sampling in cases where the subject matter, style or format of the documents has evolved over time, and the need for adequate coverage of the possibly disparate types of content in the corpus. If the facts are sparse, it may be useful to create a synthetic document with denser facts by copying relevant sentences out of multiple documents.

6.4 Generalise Your Observations

Corpus analysis alone is not sufficient to develop an effective fact extractor. Any real-world knowledge that is available to help the task should be considered; for example:-

---

[20] Consider the example of a person fact extractor. The attribute we are trying to find is "the person's name" we might represent it as one field called *name*, or choose to use three fields *firstname, initial, Surname*. Co-reference resolution based on the person's firstname will be much easier if the three field model is chosen.

- If the information required is defined by a standard (for example an e-mail address) then the standard should be consulted.

- It may also help to create a synthetic document with examples of similar text that should not match.

- If the information need is from well-structured documents, style guides for those documents may provide extra clues.

- Use common sense; for example, when developing a date fact extractor it makes sense to include all of the months of the year, even though the sample corpus might not contain any references to December.

Each of these points should be considered in conjunction with the corpus analysis.

6.5 Consider the Design

The next step is to consider the overall design of the fact extractor. It should start with checking to see if any previously developed fact extractors with a similar information extraction need might offer some insight into how to solve this task. As well as a candidate design for the primary fact extractor, this analysis may also lead to the discovery of one or more existing subordinates that may assist the task. Developer experience has shown that as fact extractors tend to be tuned to a particular corpus it is often more productive to reuse the design concepts from previously developed fact extractors as opposed to extending an existing fact extractor.

At this point it is appropriate to consider if just a single fact extractor will be developed or if the design will involve one or more subordinates. If an existing subordinate is to be reused, consider whether it will be easier to maintain it as a subordinate used by more than one composite, or if it is better to make a copy of the subordinate just for this information need. Use of a subordinate by more than one composite reduces the number of fact extractors that need to be maintained. Making a copy allows the developer to modify the subordinate for this information need or corpus, without risk of damaging the previously working composite used for a different task. The choice is normally influenced by how generic the subordinate information need is.

The next step is to consider what rules need to be developed to extract the information. The corpus analysis should have identified words that trigger the creation of a new fact. A set of create rules are developed to extract these facts. If the attributes of a single fact span more than one sentence, then a set of expand rules will need to be developed to extract these extra attributes. Expand rules will usually require the development of appropriate co-reference rules to link the partial facts together.

For each rule identified above the designer will need to develop a pattern expression that matches the text and action expressions that recover the information. Some patterns are hard to tightly define in the pattern language; instead the designer may need to employ loose patterns and then restrict the matching with appropriate guard expressions. As described in section 5.14 guard expressions can be a significant performance overhead if the pattern matches too freely.

6.6 Develop

After considering the general design of the fact extractor and any subordinates it is time to commence the development process. It will assist the future maintenance of the fact extractor if appropriate comments are added to the description section of each fact extractor explaining the intended use of the fact extractor and recording any interesting design and development decisions.

The development process starts by entering the field names that were chosen in the information need and design steps. If new (or copies of existing) subordinates were identified in the design step then fact extractors should be created for each of these and their field names should be entered. Through the use of multiple edit windows the Fact Extractor Workbench [12] supports the parallel creation of multiple fact extractors. It is a user preference to consider each identified subordinate as a separate fact extractor with its own information need or to develop the set in parallel. Experience has shown that the knowledge engineer will often oscillate between these two modes of development.

After entering the field names and initial comments, the next step is to develop the patterns that match the text. The create rules should be developed first as it is not possible to test co-reference or expand rules until the matching create rules are working. It is often easiest to try various patterns or parts of patterns in the corpus browser window and when satisfied cut and paste them into a rule.

If a need for co-reference or expand rules was identified during the design process, these should now be developed. Note that like create rules, patterns (without the `<self>`) for expand and co-reference rules can be experimented with in the corpus browser window. These patterns can include references to subordinate fact extractors.

6.7 Test

The fact extractor workbench supports an incremental design and test process. Once the initial development work has been done, the fact extractor should be tested against each document in the corpus. Inevitably examples will be found where the fact extractor does not perform as expected. In each case the knowledge engineer needs to decide whether to modify the fact extractor or accept the error. Section 2 discussed the accuracy requirements for different types of information extraction tasks.

If the decision is to modify the fact extractor then the knowledge engineer must verify that this new modification does not introduce errors where it was previously performing correctly. To do this the previous documents in the corpus should be retested. This type of testing is known as regression testing. The fact extractor workbench incorporates integrated support for regression testing. It is suggested that snippets of interesting text (perhaps at the paragraph level) be copied to a test document, and this document be used for routine regression testing. Depending on the quality requirements of the fact extractor under development, the complete corpus can be re-examined as part of a quality assurance process.

6.8 Deploy

Once the group of fact extractors for the task have been developed and tested, the knowledge engineer can issue them to the end-users. This will typically involve placing the fact extractor files in a shared folder or on an intranet website so that they can be accessed. The user needs to ensure that their fact extractor environment accesses the right folder or web location to find the new fact extractors.

6.9 Collaborative Development

To develop complex fact extractors, the developers might find that some degree of collaborative development is beneficial. This may range from an ad hoc discussion to a formal review of the proposed Fact Extractor. This collaboration should consider what constitutes a trigger to identify a new fact (i.e. the create rules) as distinct from adding attributes to an existing fact (i.e. the expand rules). The collaboration should also consider appropriate use of subordinate fact extractors.

Fact extractors can be shared amongst a workgroup by placing them on a shared file system or a web server.

7. Examples of Developing Fact Extractors

This section uses examples to explore the issues in developing real fact extractors. For each example we introduce an information need, and work through the steps introduced in section 6. Patterns and actions in this section may be spread over multiple lines for clarity or typesetting purposes, but must be entered as a single line in the fact extractor editor.

The fact extractors introduced in this section are included in the `FXLib` folder of the software distribution. Many patterns here explicitly use "`\s`" rather than a space character for clarity. Either is acceptable.

7.1 Extraction of Tightly Structured Objects

This section introduces four fact extractors to discover strictly defined facts:
- Internet Protocol Addresses
- Email Addresses
- Uniform Resource Locators
- Telephone Numbers

Many of these are driven by international standards, and consequently have tightly defined structures.

7.1.1 Internet Protocol (IP) Address

An IP address consists of a group of four number numbers, each in the range 0-255 and separated by dots. For example 123.223.111.4 .

1) **Identify** an information need: What IP numbers are referenced in a stream of communications[21]. A fact extractor can satisfy the information required with a single field called `IPNum`.

2) Collect a **relevant corpus** - a set of example documents containing the information you require, collected from the stream you will be using this fact extractor on.

3) **Examine the Corpus** to find how IP numbers are represented in these documents. They are always represented as four numbers separated by dots. There is no white space inside the IP number, and each number is up to 3 digits long, but numbers less than 100 do not have leading zeros. Consider also whether the corpus contains numbers which are not IP addresses but which match, or are similar to, the specification of an IP address.

4) **Generalise your observations** by any other knowledge, for example the largest legal value for any number is 255.

5) **Think about designing** your fact extractor. In this case, there is either an IP number, or there isn't, and all required information is contained in it. So there will be a single *Create Rule* in your fact extractor, and no *Expand Rules*. Since we are not looking for extra references to an IP number once it is found, we don't need any *Co-reference Rules*, either.

6) **Develop** your fact extractor. Write the regular expression *pattern* for your rule. A simple regular expression which matches all IP numbers is

```
[12]?[0-9]?[0-9]\.[12]?[0-9]?[0-9]\.[12]?[0-9]?[0-9]\.[12]?[0-9]?[0-9]
```

This *pattern* will also accept a few strings which are not valid IP numbers, if an octet's value is between 256 and 299 inclusive. For performance reasons, you may be prepared to accept these rare *false positives*.

Check that the pattern is matching what you expect it to match, and then write the actions to go with it. In this case, the action is very simple, and is simply

```
IPNum = MATCH("0")
```

7) **Test** your fact extractor. In this example it will help to create a test document that has a range of valid and invalid IP addresses. Remember to test IP addresses at the beginning, middle and ends of sentences. After the basic behaviour has been confirmed continue testing on a range of real documents.

Further considerations: If the *false positive* matches are not acceptable, you need to use parentheses to separate out each group, and use a *guard expression* to test each *match group* for being in the right range. This could result in a complex *guard expression*. It

---

[21] A related information need would be to reason about subnets. In this case each octet might be extracted to a separate field.

may turn out to be simpler to create a *subordinate fact extractor* to find each octet. In that case, the main fact extractor pattern would become "(<octet>)\.(<octet>)\.(<octet>)\.(<octet>)". The pattern of octet should be [12]?[0-9]?[0-9] with a guard expression such as LESSTHAN(MATCH("0"), "256") to ensure the number is in the right range. Note that LESSTHAN was not a standard function in earlier releases of the Fact Extractor System.

In the future, this fact extractor may need to be revisited to upgrade it to recognise IP version 6 addresses as well as the older version 4 addresses.

7.1.2 Email Address

1) Identify an **information need**: Internet (SMTP) email addresses.

Email can also be sent by other protocols such as X.400, Lotus Notes or Microsoft Exchange which have different address formats and these are not considered further in this example. The information can be represented by a single field called address.

2) Collect a **relevant corpus** - a set of example documents containing the required information, collected from the stream on which this fact extractor will be used. Even people who use other email systems actually quote their internet-style address in documents, so that is the only style of address required of this fact extractor.

3) **Examine** the Corpus to find how email addresses are represented in these documents. They are always represented as <user>@<host or domain>. Sometimes they are next to "mailto:" or contained in "<…>", but not always. They never have spaces.

4) **Generalise** the observations by any other relevant knowledge: This might include checking the standards (RFC822 and its successors) for legal characters and for the complete list of Top-Level Domains (TLDs).

5) Think about **designing** the Fact Extractor. In this case, an email address is a sequence of letters and similar characters, an '@', and another sequence of characters and dots. So there will be a single Create Rule in the Fact Extractor.

6) **Write** the regular expression pattern for the rule. A simple regular expression which matches these addresses is

```
[A-Za-z0-9\.&\-_]+@[A-Za-z0-9\.\-]+\.[A-Za-z]+
```
This pattern does not attempt to ensure that the string ends with a valid TLD.

7) **Test** the FX. Verify that it is finding expected addresses, and none that are not.

Further considerations: If there is a requirement to ensure that strings are only accepted if they end with a real TLD, make a subordinate TLD Fact Extractor (probably as a list), and then replace the last '[A-Za-z]+' with '(<TLD>)'. Another consideration might be to add a DisplayName field to the fact, and attempt to populate it with the human-readable name of the person who uses that address. Often this is provided near the email address, but not always.

7.1.3 URL

A Uniform Resource Locator (URL)[22] provides a reference to a resource. The most common examples are references to web pages such as http://mypage.com.au but there are many other kinds: for example ftp://myhost/mydocument.doc. This example only considers URLs in documents not URLs behind anchors in web pages. Parameters following a URL are not considered part of the URL.[23]

1) Identify an **information need**: Recover URLs in documents. For this example there are three fields, `protocol`, `Hostname` and `URL`. At this point consider whether the need is to be sure to only collect valid URLs or if it is to collect patterns that are similar to URLs. For example is the fact extractor required to discover text that the author intended to be a URL but, perhaps due to poor typing, is not actually a valid URL? In most cases it will be appropriate to leave strong checking to a subsequent processing step.

2) Collect a **relevant corpus** - a set of example documents containing the required information, collected from the data source that will be used by this fact extractor. This would probably identify whether HTML is important, or only plain text.

As fact extractors are designed to operate on English text independently of the file format or data representation, HTML markup is usually ignored by the fact extractor system apart from using it to identify implicit sentence breaks through paragraph block analysis. In this particular example, it is possible that the URLs of interest are part of that markup, so fact extractor users might wish to process HTML files as plain text files for this purpose. Information on how to tell the fact extractor to treat files in different ways is included in the online help of the Fact Extractor Workbench [12].

3) **Examine** the Corpus to find how URL's are represented in these documents. It is quickly noticed that there is a big difference in the way URL's can be described in HTML files compared to plain text documents. For this example it is assumed that the documents are plain text. Sometimes when the protocol is HTTP it is omitted. For example http://www.mypage.au is frequently abbreviated to www.mypage.au.

4) **Generalise** the observations by any other knowledge: URL's are defined by standards so it is sensible to consult these standards.

5) Think about **designing** your Fact Extractor. In this case the general form of a URL is <scheme>:<scheme specific part> so it makes sense to have a create rule for each scheme that you care about plus an extra rule to deal with examples where the scheme is omitted. An alternate approach is to create a subordinate for each URL scheme and a primary fact extractor that integrates the subordinates. The second approach may assist fact extractor re-use and assist subsequent maintenance.

---

[22] Note that URL is an informal term associated with popular Uniform Resource Identifier (URI) schemes; see the World Wide Web Consortium for a discussion on addressing schemes[15].

[23] In fact the current implementation of fact extractors are unable to extract the parameters following the "?" in a URL. The default sentence processing considers a question mark to signify an end of sentence and effectively breaks the URL. It is expected that a future release of the fact extractor system will allow some degree of user tailoring of the sentence breaker to address this and related issues.

6) For each scheme to be implemented, **write** the regular expression pattern for the rule. The following rule matches a simple HTTP URL.

```
Pattern: http://([_a-zA-Z0-9\.-]+)(/[_a-zA-Z0-9\.-]*)*
Action    protocol = "HTTP"
          URL = MATCH("0")
          Hostname = MATCH("1")
```

The next rule matches a simple HTTP URL that is missing the formal "http://". It is inferred that this is a HTTP URL from the common practice of starting the hostname with "www".

```
Pattern: (www\.[_a-zA-Z0-9\.-]+)(/[_a-zA-Z0-9\.-]*)*
```
The actions are similar to the first rule.

A third pattern collects FTP-based URLs. The first grouping deals with FTP requests that contain a user name.

```
Pattern :  ftp://([_a-zA-Z0-9\.-]+@)?([_a-zA-Z0-9\.-]+)
           (/[_a-zA-Z0-9\.-]*)*
```
Hostname is now the second match otherwise the actions are similar to the first rule.

7) **Test** the FX. In this example it makes sense to construct a document that has a wide set of legal (and illegal) URLs. After the basic behaviour of the fact extractor has been confirmed, continue testing on a range of real documents.

Further considerations: A fully developed fact extractor should consider the implications of Universal Resource Identifiers as opposed to Universal Resource Locators.

7.1.4 Phone Numbers

1) Identify an **information need**: Discover telephone numbers. Telephone numbers can be for mobile or landline telephones, and may include area codes and/or country codes. They can also be represented in different ways. Ideally, the fact extractor should be designed to discover all of these. The information need can be met with a single field, `Number`.

2) Collect a **relevant corpus:** A set of example documents containing the required information, collected from the intended data stream such as emails, news stories, company reports, etc. Whatever the source, the corpus should be a representative set of documents used for developing and subsequently testing the fact extractor.

3) **Examine** the corpus to find out how telephone numbers are represented in it.

4) **Generalise** the observations by any other knowledge: If the corpus uses only a limited set of ways of representing telephone numbers, include additional commonly used representations to make the fact extractor more generic. Some ways that

telephone numbers are represented include: +61 8 8259 1234, (08) 8259 1234, 08 8259 1234, 8259 1234, 82591234, 08-8259-1234, x91234, 0409 123 456

5) Think about **designing** the Fact Extractor: One way of categorising representations for telephone numbers may be as follows:

- Local numbers
- Numbers with country and area codes
- Numbers with area codes
- Mobile numbers
- 1 800 and 1 300 numbers
- 13 xxxx numbers
- 5-digit extensions

Using this categorisation scheme, seven rules were defined, one for each category. An alternative design would have been to use a subordinate fact extractor for each of the categories of telephone numbers and put them together in the top-level fact extractor.

6) For each category of telephone number, **write** the pattern and action for the rule. Patterns and actions for the seven create rules are detailed below.

Rule1 finds local phone numbers of the form: xxxx xxxx or xxxx-xxxx. The pattern for the rule is divided into two groups; the first locates the first four digits of the number, followed by an optional space or hyphen, followed by the second group which locates the next four digits of the number. The \s regular expression term is used to match white-space characters including space, tab or newline. The pattern and action for this rule are:

```
Pattern: (\b([0-9]{4})[\s-]?[0-9]{4})
Action: Number = MATCH("1")
```

Rule2 finds international phone numbers with a country and area code. It can match phone numbers of the form +xxx xxxx xxxx, and the US forms: +1(xxx) xxx xxxx or +1-xxx-xxx-xxxx. The embedded spaces in the phone number are optional, the pattern and action for this rule are:

```
Pattern: ((\+(([0-9]){3}[\s-]?)(([0-9]){4}[\s-]?){2})|
         (\+1\(?[0-9]{3}\)?[\s-]?[0-9]{3}[\s-]?[0-9]{4})))
Action: Number = MATCH("1")
```

Rule3 finds phone numbers with an area code. It can match phone numbers of the forms: (08) xxxx xxxx or 08-xxxx-xxxx. The parentheses around the area code and any embedded space or hyphen are optional. The pattern and action for this rule are:

```
Pattern: ((\(?([0-9]){2}\)?)[\s-]?(([0-9]){4}[\s-]?)([0-9]){4})
Action: Number = MATCH("1")
```

Rule4 finds a mobile phone number. It can match phone numbers of the form xxxx xxx xxx, and allows for an optional embedded space or hyphen. The pattern and action for this rule are:

```
Pattern: (((([0-9]){4})([\s-]?([0-9]){3}){2})
Actions: Number = MATCH("1")
```

A telecommunications regulatory authority could have been consulted for specific information on the allowable first four digits of mobile numbers, to ensure better accuracy. However, to keep the example simple it accepts any four digits.

Rule5 finds 1-800 or 1-300 phone numbers. It can match phone numbers of the forms: 1-800-xxx-xxx or 1 300 xxx xxx, and allows for an optional embedded space or hyphen. The pattern and action for this rule are:

```
Pattern: \b(1[\s-]?[38]00([\s-]?([0-9]){3}){2})
Action: Number = MATCH("1")
```

Rule6 finds a 13-xxxx phone number. It can match phone numbers of the forms: 13x xxx or 13 xx xx. It allows for an optional embedded space or hyphen between the groupings of digits in both forms. The pattern and action for this rule are:

```
Pattern: (13[\s-]?(((([0-9][0-9][\s-]?){2})|
         ([0-9][\s-]?[0-9]{3})))\b
Actions: Number = MATCH("1")
```

Rule7 finds 5-digit telephone extensions in various forms, including: xzzzzz, Xn zzzzz, on zzzzz, Extn zzzzz, ph: zzzzz. The pattern for this rule looks for an optional prefixed string matching the first parenthesised group, followed by a set of 5 digits representing the telephone extension in the second. The pattern and action for this rule are as follows:

```
Pattern: \b([Oo]n|[Pp]h:?|[Extn]|[Xx]n|X|x)?\s?
         (([0-9]){5})\b
Actions: Number = MATCH("2")
         matchrange = "2"
```

Rules with more complex regular expressions appear before rules for the simpler ones in this example. This is coincidental as all create rules are evaluated and the fact discovered is the one corresponding to the rule with the longest match (see section 4.4.3 for details on processing create rules).

7) **Test** the fact extractor. Verify that it is finding all the expected telephone numbers and none that are not. For testing the fact extractor, it might be helpful to create a sample document with telephone numbers represented in all the expected formats. For completeness the document should also include other reasonable groupings of numbers that are not intended to be discovered as telephone numbers.

7.2 Dates and Times

This next section considers two temporal fact extractors: dates and times.

7.2.1 Dates

1) Identify an **information need**: For this example the information need is to find dates in a document. This could be represented by a single field called `ISODate` that conforms to the ISO specification for dates; however this example uses a three field representation consisting of day, month and year.

2) Collect a **relevant corpus** - a set of example documents containing dates, collected from the data source this fact extractor will be used on.

3) **Examine** the corpus: dates are represented in many ways, for example:

> 12/8/03
> 8/12/03
> 2003-08-12
> 20030812
> 12 Aug 2003
> 12 August 2003
> August 12, 2003
> Yesterday, tomorrow
> 120000Z Nov 2003 (Military Date-time group)

4) **Generalise** your observations by any other knowledge: In this example apply some common sense: use the knowledge that there are twelve months in a year and a defined number of days in each month. The names and likely abbreviations of the months are also known.

5) Think about **designing** the Fact Extractor. A tractable approach is to have a create rule for each kind of date representation. To help with the maintenance of the fact extractor it is good practice to name the rules in a way that conveys the pattern they are targeting, for example "dd mm yy" or "dd month yyyy". This example uses a subordinate fact extractor (`DEF_Month`) to recognise and handle month names and abbreviations. The design should consider how to handle partial dates as documents might not always fully qualify dates. They may be newspaper articles where the year is derived from the beginning of the article or the date of publication. A separate publication date fact extractor may be required to capture these details, which may be accessed from the actions of the date fact extractor. Unfortunately, these sorts of extensions are usually closely related to the document corpus, and as such cannot be adequately addressed here. This example attempts to resolve dates that are lacking a year, but no other partial dates.

6) **Write** a pattern for each rule. Test each pattern against valid values and invalid values. It is of assistance to manually produce a test document that has a collection of example dates and non-dates. When the patterns successfully match the dates complete the actions to extract the information and test again.

The first rule catches dates containing day, month and year in that order, with spaces or common separators between the parts. A range of separators (white-space, '.', '/'and '-') is accepted (subgroup 2), and a back-reference[24] (\2) is used to ensure that the second separator is the same as the first one.

```
Rule: Day/month/year
Pattern: \b(0?[1-9]|[12][0-9]|3[01])(\s?[\s./-]\s?)(1[0-2]|0?[1-
9])\2(([12][0-9])?[0-9]{2})\b
Actions: Year = MATCH("4")
        Day = MATCH("1")
        Month = MATCH("3")
```

There is a similar rule for detecting American-style notations putting the month first. Switching the order of these two rules in the fact extractor changes which one is preferred if both match, as can happen in the first 12 days of each month.

```
Rule:    Month Day Year (American)
Pattern: \b(1[0-2]|0?[1-9])(\s?[\s./-]\s?)(0?[1-9]|[12][0-
9]|3[01])\2(([12][0-9])?[0-9]{2})\b
Actions: Year = MATCH("4")
        Day = MATCH("3")
        Month = MATCH("1")
```

A Military Date-time group (DTG) rule recognises dates specified as part of a DTG. These have the day of month, then the time (with time zone), usually followed by the month and year, although these are sometimes left out where the author believes they are obvious from the context. There is usually no space between the date and time parts, although one source of operational reports in the sample corpus consistently used a space, so this rule accepts an optional space.

```
Rule: Military Date-Time Group (DTG)
Pattern: ([0-3][0-9])\s?([0-2][0-9])([0-5][0-9])[A-Z]
(<DEF_Month>) (([12][0-9])?[0-9]{2})
Actions: Day = MATCH("1")
        Month = FIELD("4", "asDigit")
        Year = MATCH("5")
```

The next two rules handle dates with only a day and month, as the year can usually be guessed from an earlier reference. This requires two rules to handle both day/month and month/day forms. The CACHE() function is used to get the year from the most recently found date.. This may not be appropriate if the corpus frequently talks about annual events, such as "Anzac Day is on April 25", because it will choose some year which might not be useful.

```
Rule: Day Month, guess year
Pattern: \b(0?[1-9]|[12][0-9]|3[01])(\s)(<DEF_Month>)\b
Actions: Day = MATCH("1")
        Month = FIELD("3", "asDigit")
```

---

[24] Back-references are a regular expression concept used here to ensure that the same symbol is used between month and year as is used between day and month. This avoids constructs such as 02/07-23 being recognised as a date.

```
              Year = CACHE("DEF_Date", "Year", "unknown")
Rule: Month Day, guess year
Pattern: \b(<DEF_Month>)(\s)(0?[1-9]|[12][0-9]|3[01])\b
Actions: Day = MATCH("3")
         Month = FIELD("1", "asDigit")
         Year = CACHE("DEF_Date", "Year", "unknown")
```

ISO Date formats are fixed lengths. The year always occupies 4 digits; month and day are always two digits. For human reading, the fields are separated by dashes, but these are optional for machine processing. This rule again uses a back-reference (\2) to ensure that if one dash is present, they both are.

```
Rule:    ISO Date format
Pattern: \b([12][0-9]{3})(-?)([01][0-9])\2([0-3][0-9])
Actions: Month = MATCH("3")
         Day = MATCH("4")
         Year = MATCH("1")
```

Other common formats for writing dates are January 12th, 2004 and 12th of January 2004.

```
Rule:    month day, year
Pattern: (<DEF_Month>)(\s?)(0?[1-9]|[12][0-9]|3[01])
         (st|nd|rd|th)?,\s(([12][0-9])?[0-9]{2})
Actions: Month = FIELD("1", "asDigit")
         Day = MATCH("3")
         Year = MATCH("5")

Rule:    ordinal of month, year
Pattern: \b(0?[1-9]|[12][0-9]|3[01])(st|nd|rd|th)?(\sof)
         ?\s(<DEF_Month>),?\s(([12][0-9])?[0-9]{2})\b
Actions: Day = MATCH("1")
         Month = FIELD("4", "asDigit")
         Year = MATCH("5")
```

Some log-based documents represent dates as Month Day, time, year.

```
Rule:    month, day, time, year
Pattern: \b(<DEF_Month>)(\s)(0?[1-9]|[12][0-9]|3[01])\s
         ([0-2][0-9]:[0-5][0-9]:[0-5][0-9])\s
         (([12][0-9])?[0-9]{2})
Day = MATCH("3")
Month = FIELD("1", "asDigit")
Year = MATCH("5")
```

Relative forms of date such as "yesterday", "today", "tomorrow", and "three days ago" cannot be readily included in a generic Date fact extractor. They are highly sensitive to the particular corpus or document source being analysed. This is because they are usually relative to the date of the report (or the date someone was quoted), rather than to any surrounding dates in the text. As such, there usually needs to be a special "report date" fact extractor written to recognise the date of the report, and then the

`CACHE()` function can be used to refer back to the "report date" fact from which relative dates may be calculated.

This fact extractor sometimes reports single digit days and months with a leading zero. The fact extractor software distribution includes DEF_Date with more complex actions to trim out the leading zeros.

The fact extractor is tolerant of 2-digit years. In practice, all dates may be required to be reported with 4-digit years. This requires a decision about which 100-year window the 2-digit years should be assumed to be in. The following action assumes two-digit years in the range 1950-2049:

```
Action:  Year = SUB(IF(LESSTHAN(MATCH("5"), "50"),
                    ADD("2000", MATCH("5")),
                    IF(LESSTHAN(MATCH("5"), "100"),
                       ADD("1900", MATCH("5")),
                       MATCH("5"))),
                "0", "4")
```

7.2.2 Times

1) Identify an **information need**: In this example the need is to identify a range of references to times in a document. While it is possible to consider the need for well specified times to be recovered from semi-structured files, like a log file, this example is concerned with general references to times in free text. As it is unlikely that times by themselves would ever be a sensible information need, this fact extractor is intended for use with other fact extractors. Perhaps it could be part of a loose collection in the FormFiller environment or used more tightly as a part of a composite fact extractor. As a result a fair degree of over generation (false matches) can be tolerated. For this example it was decided to collect time as hours and minutes on a twelve-hour clock. An AM/PM flag describes the period and an optional time zone indicator could also be included. The fields are named `Hour`, `Min` and `Meridian`.

2) Collect a **relevant corpus** - a set of example documents containing the information you require, collected from the data stream this fact extractor will be used on.

3) **Examination** of the corpus reveals that times appear in a wide variety of ways, for example:

16:44
16:44:56
4:45 PM
07:15Z
1644+0930
10 minutes ago
now

As the representation of these times is independent of actual usage in real text, it is of benefit to construct a test document with a wide range of sample times interspersed

within some dummy text. This test document was used to speed up the initial development of the fact extractor; real documents should be used for final testing.

4) **Generalise** your observations with any other knowledge: Clearly the numeric representation of a time is quite well constrained. We know that there are two twelve-hour periods or twenty-four hours in a day and sixty minutes in an hour but the textual representation of a time can be quite varied. For this example only times represented as a collection of digits are found.

5) Think about **designing** the Fact Extractor: This fact extractor uses a straightforward collection of create rules; co-reference and expand rules are not necessary.

6) **Write** the pattern and action rules. All of the rules are fairly similar so only the first one is explored here. The first rule looks for an hours-minutes-seconds string. There is nothing unusual in the pattern. A guard expression is used to eliminate some obvious false positives by rejecting results that would lead to an invalid time. The first and third actions handle hours greater than twelve and convert them back to a twelve hour clock.

```
Rule: hh24:mm:ss
Pattern: (([0-2][0-9])[\.:]?([0-6][0-9])[\.:]?([0-6][0-9]))
Guard: AND( LESSTHAN( MATCH("2"), "24"),
            LESSTHAN( MATCH("3"), "60"),
            LESSTHAN( MATCH("4"), "60"))
Actions:
    Hour = IF(GREATERTHAN(MATCH("2"), "12"),
              FORMATNUMBER(SUBTRACT(MATCH("2"), "12"), "I2"),
              MATCH("2"))
    Min = MATCH("3")
    Meridian = IF(GREATERTHAN(MATCH("2"), "11"), "PM", "AM")
```

The other create rules deal with variations on the possible representation of a time. In cases where more than one create rule could match the text the largest rule is used, see section 4.4.3.

7.3 Names of People

This section introduces fact extractors that look for names of people (or organisations). There are two fundamentally different approaches to this problem depending on whether or not the set of names to be found is known.

7.3.1 Names from a List of Known Names

1) Identify an **information need**: For this example find names of people in a document from a pre-existing collection of names. This information is represented by a single field `name`.

2) Collect a **relevant corpus** - a set of example documents containing the required information, collected from the data stream the fact extractor will be used on. In this

example the corpus is less important than usual as the list of required names is defined elsewhere. The primary value of the corpus is to determine if the rate of false positives can be tolerated.

3) **Examine** the corpus. It will most likely contain examples of the names used. There might also be strings that are the same as a wanted name but are not actually a name. There may also be examples of co-references like pronouns and name abbreviations.

4) **Generalise** your observations by any other knowledge you have. For a simple list of known names, there is nothing extra to add here.

5) Think about **designing** the Fact Extractor. The obvious way to do this task is to create a list fact extractor made up from the list of wanted names.

6) **Write** the list entries for the fact extractor. This can be done from within the fact extractor development environment but it may be easier to capture the list of names directly from other sources such as a database. The format for a list fact extractor is one list item per line in a text file called <listname>.fx. An excerpt from the example list fact extractor, "DEF_BaliBombingNames.fx" follows:

```
…
fikiruddin
fikiruddin muqti
fuad amsyari
hafiz ismael
haji acing
haji aceng suheri
haji ismail pranoto
haji khoir affandi
haji mansur
…
```

Notice that all the list items are in lower case. This allows the fact extractor to discover matching list items in lower, upper as well as title case. If list items are written in mixed or upper case, case-sensitive matching will occur. The information need can help to decide which option is appropriate for your particular list fact extractor(s).

Further considerations: If this simple list fact extractor generates too many false matches, or the information need requires capturing pronouns and abbreviations then the use of a composite fact extractor is recommended. The composite fact extractor would invoke the simple list fact extractor and also manage over-generation and co-reference resolution.

7.3.2 Unbounded Names

1) Identify an **information need**: In this example the task is to extract words from the text that represent names of people without knowing in advance the set of names. In addition, it is required to pick up as many co-references as possible. This is intended to be a low-precision tool which would be used to assist a human operator. It is not be expected to be precise enough for automatic operation. The only field required is `fullname`.

2) Collect a **relevant corpus** - a set of example documents containing names, collected from similar sources to what the fact extractor will be used on.

3) **Examine** the corpus. In an ideal world, the corpus would always introduce people the first time with title, first name, and surname (such as Mr. Scott Davis), allowing quite high accuracy. Unfortunately, this is not common in many styles of document. For the purpose of this fact extractor, that information will be used if available, but in this corpus, people are usually identified initially by first name and surname, and then referred to just by their first name. The fact extractor looks for names based on the capitalisation of words, which is prone to finding extra things which are not people's names.

4) **Generalise** your observations by any additional knowledge: There are a number of common titles used to address people, such as Mr, Mrs and Miss. The fullstops after Mr and Mrs are frequently (but not always) left out in modern informal writing.

5) Think about **designing** the Fact Extractor. In this case, the use of co-reference rules to match parts of the name (such as first name) requires the facts to have separate fields for title, first name and surname. These fields will be filled in so that the full name field contains a complete name (including title and middle names if provided), first name and surname get a single word each. It is assumed that the last word in a person's name is their surname, even though this is not always true in some cultures.

6) **Write** the fact extractor. The first create rule is for title, first name, last name and any middle names provided.

```
Pattern: (Mr|Mrs|Miss|Dr|Doctor|Sir|Dame)\.?\s
         ([A-Z][a-z]+)(\s([A-Z][a-z]+))+
Actions: title = MATCH("1")
         firstname = MATCH("2")
         surname = MATCH("4")
         fullname = MATCH("0")
```

There may be quite a bit of effort in getting these regular expressions right. The ability to test patterns in the corpus browser window is invaluable in experimenting with spaces and parentheses to get a regular expression which accurately matches the representation of names in your corpus. The pattern above matches names which start with a title (with an optional fullstop), then have a first name and any number of other names. The last capitalised word is used as the surname. It may appear odd that the middle names (if any) are matched as earlier surnames, rather than as extra first names. This allows the pattern to be a little simpler than if they were explicitly matched as middle names. As group 3 is a repeating group containing group 4, MATCH("4") only returns the last value matched by group 4. The first name needs to be kept separate to fill the `firstname` field.

The next create rule is for the more common "firstname surname" form. It is just a simpler version of the first rule, and also handles middle names.

```
Pattern: ([A-Z][a-z]+)(\s([A-Z][a-z]+))+
Actions: firstname = MATCH("1")
         surname = MATCH("3")
```

```
                    fullname = MATCH("0")
```

However, this pattern also matches text where a title is used without a fullstop, in which case the title would be erroneously treated as the firstname. The problem may be avoided by forcing a negative lookahead ("(?!…)") for titles before the first name. The word "The" is also excluded as it indicates organisations not people:

```
Pattern: (?!Mr|Mrs|Miss|Dr|Doctor|Sir|Dame|The)
         ([A-Z][a-z]+)(\s([A-Z][a-z]+))+
Actions: firstname = MATCH("1")
         surname = MATCH("3")
         fullname = MATCH("0")
```

A third create rule is used to collect title, initials and last name. The title is optional in this rule, but a word break at the beginning is enforced to ensure that an initial is not picked up at the end of an acronym preceding a capitalised word.

```
Pattern: \b((Mr|Mrs|Miss|Dr|Doctor|Sir|Dame)\.?\s)?
         (([A-Z]\.?)\s)+([A-Z][a-z]+))+([A-Z][a-z]+)
Actions: title = MATCH("1")
         surname = MATCH("5")
         fullname = MATCH("0")
```

A co-reference rule is used to detect reuse of the first name only:
```
Pattern: [A-Z][a-z]+
Guard expression: EQUAL(MATCH("0"), VALUE("firstname"))
```

A co-reference rule to detect formal references of the form Mr Davis:

```
Pattern: (Mr|Mrs|Miss|Dr|Doctor|Sir|Dame)\.?\s([A-Z][a-z]+)
Guard expression: AND(EQUAL(MATCH("1"), VALUE("title")),
                      EQUAL(MATCH("2"), VALUE("surname")))
```

7.4 Placenames

Placenames may be found in a number of ways. First is looking for a list of known places where any reference to them is interesting (the same as people's names in section 7.3.1). The simplest way to do that is just to have a list fact extractor that contains all the interesting placenames. This is useful if some later process will plot them on a map based on the names, or if the fact extractor is only being used as a filter to highlight documents that might need to be read fully by someone. Some military messages frequently contain a grid reference or geographic coordinates near a place name, so a fact extractor could extract those as well. In other text such as news reports, a place name can only be identified from its context, for example if something happens "...at Mumble...", then Mumble is probably a placename (or a time!).

1) Identify an **information need**: In this case, this fact extractor was developed for a JWID 2004[25] service to identify place names so a gazetteer service can be used to

---

[25] JWID, the Joint Warrior Interoperability Demonstration will be known as CWID (Coalition …) from 2005.

display them on a map. Where possible, it also identifies the type of place that the name might represent.

2) Collect a **relevant corpus** - a set of example documents containing the information you require, collected from the stream you will be using this fact extractor on. As the details of the JWID scenario were not yet available, the corpus used for developing the fact extractors was mostly drawn from internet web sites showing current news reports. To ensure that the fact extractors were developed in a generic manner, a range of different news web sites was used.

3) **Examine** the corpus. Looking at the corpus of news articles, it becomes difficult to determine how even a human decides whether a word or group of words should be considered as a place name. One indicator seems to be that a small group of prepositions frequently appear before placenames, and that the words in a placename are usually capitalised.

4) **Generalise** your observations by any other relevant knowledge. ISO 3166-1[11] provides a list of official country names and codes.

5) Think about **designing** your Fact Extractor. The `JWID_places` fact extractor contains the names and place types of the fictitious places created for JWID 2004. The `ISO_Country_Names` list fact extractor contains country names. A composite `DEF_placename` fact extractor uses these subordinate fact extractors, as well as rules using prepositions and other common aspects of the text.

6) **Write** the Fact Extractors. The most common indicator of placenames seems to be a preposition followed by a group of capitalised words. The most common placename-introducing prepositions are "at", "in" and "to".

```
Rule: at|in|to …
Pattern: \b(at|in|to|near|from)\s(([A-Z][a-z]+)(\s[A-Z][a-z]+)*)
Actions: matchrange = "2"
         name = MATCH("2")
```

A country names list fact extractor was constructed by downloading the file at http://www.iso.ch/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/list-en1-semic.txt then extracting the official names using the Unix command:

```
awk -F\; '{print $1}' list-en1-semic.txt | tr '[:upper:]'
'[:lower:]' > FXLib/ISO_Country_names.fx
```
and deleting the preamble. This allows a second rule

```
Rule: countrynames
Pattern: (<ISO_Country_names>)
Actions: placename = MATCH("0")
         Placetype = "Country"
```

Even if a place is not introduced by the common prepositions, if it is prefixed by a compass direction, it is usually a placename.

```
Rule: compass
Pattern: (North|South|East|West|Central)(ern)?(\s[A-Z][a-z]+)+
Actions: placename = MATCH("0")
```

Corpus analysis also demonstrated that frequently a named place was used to identify a particular facility or event at that place. The word "the" turned out to be quite good at identifying this use. It does introduce some false positive matches, but also picks up a number of placenames that would otherwise have been missed.

```
Rule: the ...
Pattern: [Tt]he\s(([A-Z][a-z]+)(\s([A-Z][a-z]+))*)
Actions: placename = MATCH("1")
         matchrange = "1"
```

7.5 Relationships

7.5.1 Person Names and Aliases

1) This example is based on a real **information need** to find names and aliases associated with Jemaah Islamiah from a lengthy public domain document. The document also contains multiple languages, making it tedious for a non-linguist to process. The extracted information is intended for input into data visualisation tools such as those used to discover social networks. Normally it would not be appropriate to write a fact extractor for a single document, but in this case fact extractors are used to save time and tedium. The fact extractor produces (<name>,<alias>) pairs. The information need required a high level of accuracy.

2) The **corpus** is a 92 page text document converted from its original PDF format. While the current version of the Fact Extractor System provides experimental support for simple PDF documents, the conversion was necessary as the PDF document was multi-columned, and was not appropriately handled by the Fact Extractor system.

 3) A cursory **examination** of the document revealed that some of the common phrases used to indicate the presence of aliases are:

  a. <name> alias <alias>
  b. <name>(a.k.a. <alias>)
  c. <name> who also used the aliases: <alias>,…
  d. <name> has a host of aliases: <alias>,…
  e. <name> had many aliases: <alias>,…

4) It is useful to **generalise** the fact extractor to also detect aliases in other similar documents. Another phrase for identifying aliases is:

  f. <name> who went by the aliases <alias>, …

As this list contains both present ("has") and past ("used", "had") indications of aliases, the planned fact extractor might also be generalised to allow any of these words interchangeably by a pattern part like "(ha|use)[sd]" rather than just one of those words. The same issue might occur for "(went|goes)".

5) The fact extractor **design** was based on the six phrases identified in steps 3 and 4 to discover names of people who had one or more aliases. The information need required

facts to be output as (<name>, <alias>) pairs. Where a person had more than one alias, multiple facts were required. This task required a high degree of accuracy and it was decided that a single fact extractor could not meet this criteria. The solution chosen was to develop two fact extractors for use in the FormFiller[6] application under manual control. The two fact extractors are described below:

a) `DEF_UnboundEntity` is designed to discover names of people, and has a single field - `name`. It uses clues in the text such as capitalisation, and consequently picks up other proper nouns such as place names or month names. This over-generation of facts is tolerated as it is intended to be used interactively together with `DEF_AliasedPerson`.

b) `DEF_AliasedPerson` discovers names of people with aliases, and also has a single field – `name`. It is important for this fact extractor to discover all persons who have one or more aliases.

The FormFiller [6] application allows both fact extractors to be applied simultaneously merging facts discovered from both into a single record, under user control.

`DEF_AliasedPerson` accurately discovers each person who has an alias, and provides a value for the <name> output element. `DEF_UnboundEntity` over-generates facts by discovering person and other names. However, by focusing on names that appear in the vicinity of where `DEF_AliasedPerson` finds names, it is possible for the user to easily identify those names that are actually aliases. The FormFiller application supports the easy creation of multiple records when a person has more than one alias.

6) For each of the fact extractors described in the design phase a set of rules were developed and **written** as follows.

a) `DEF_UnboundEntity` has a single rule to discover named entities such as person, place and organisation names. It uses title case to discover proper names, and utilises a combination of look-behind and guard expression to target person names more specifically. It allows for a name to optionally contain the string "bin", "al", "di" or "van".

Over-generation is controlled by using negative look-behind (beginning with (`"(?<!…)"` in the first group) in the pattern, and a guard expression. The negative look-behind is used to discover title case words not preceded by words like "than", "an", "in", a single digit number from 0 to 9, and so on. The guard expression is used to enumerate a stop-word list derived from examining the corpus.

The rule also uses `matchrange` to set the range of the fact.

The rule specification for this fact extractor is:

```
Rule: entity name
Pattern:  (?<!(an|[Ii]n|\s[0-9]|\sa|he|on|en|om|'s)\s)\b
          (([A-Z][a-'\.]+)
           (\s([Bb]in|[Aa]l|[Dd]i|[Vv]an)-?\b)?
```

```
                (\s?[A-Z][a-z']+){0,4}),?
Guard Expression:
   NOT(OR(EQUAL(MATCH("2"),"The"),
          EQUAL(MATCH("2"),"See"),
          EQUAL(MATCH("2"),"But"),
          EQUAL(MATCH("2"),"Just"),
          EQUAL(MATCH("2"),"Key"),
          EQUAL(MATCH("2"),"This"),
          EQUAL(MATCH("2"),"Indeed"),
          EQUAL(MATCH("2"),"And"),
          EQUAL(MATCH("2"),"As"),
          EQUAL(MATCH("2"),"They"),
          EQUAL(MATCH("2"),"After")))
Actions:  name = MATCH("2")
          matchrange = "2"
```

b) `DEF_AliasedPerson` has three rules. Rule 1 is based on phrase **a** on page 43; rule 2 is based on phrase **b**; and rule 3 combines phrases **c–f**. All rules invoke `DEF_UnboundEntity` as a subordinate fact extractor.

The first group in all the rules, `(<DEF_UnboundEntity>)`, finds candidate names. This is followed by phrases indicating that the entity found has one or more aliases. The rules use the special field `matchrange` to restrict the range of the discovered fact. This is useful for example when a phrase like "A alias B alias C" occur in the text. In this case B and C are both aliases for A. Restricting the range of `DEF_AliasedPerson` to A's range allows `DEF_UnboundEntity` to then pick up B and C as aliases.

```
Rule1: name alias name

Pattern: (<DEF_UnboundEntity>),? alias (<DEF_UnboundEntity>)

Actions:   name = MATCH("1")
           matchrange = "1"

Rule2: name a.k.a.

Pattern: (<DEF_UnboundEntity>)\s\(a\.k\.a\.

Actions:   name = MATCH("1")
           matchrange = "1"

Rule3: name who also used the aliases
Pattern:(<DEF_UnboundEntity>),?\s(who also used the|who
        also went by the|had many|has a host of)\s
        alias(es)?
Actions:   name = MATCH("1")
           matchrange = "1"
```

7) **Test** and develop these fact extractors iteratively.

7.5.2 Relationships

Identify an **information need:** Understanding relationships between people and other entities is a necessary step towards revealing social networks and otherwise hidden loyalties between people who might appear to be unconnected.

1) Collect a **relevant corpus:** in this example fact extractors are likely to be highly targeted to a particular information source. This may include writing slightly different fact extractors (or rules) for each different newspaper website being monitored. So instead of developing an actual fact extractor we will just explore the issues.

2) E**xamine** the corpus: The information need leads to the understanding that there are entities and relationships between those entities. The corpus must be studied to find out how these relationships are described.

3) **Generalise** the observations by any other relevant knowledge. For example if a document says "brother" or "father", make sure that "sister" or "mother" are also found.

4) Think about **designing** the Fact Extractor. The corpus analysis suggests the need for two types of fact extractors:

   a) One (or more) subordinate fact extractors that look for named entities, perhaps people and organisations; and

   b) Fact extractor(s) that look for interesting relationships between named entities. This may be either a composite fact extractor that uses the entity fact extractors, or more subordinates that are combined later (either in a mega-composite, or interactively using the FormFiller tool as in section 7.5.1). For example we might require a family-relationships fact extractor and a membership or business relationships fact extractor.

   In a complex composite fact extractor, it is usually better to have the composite selecting relevant facts from a slightly bigger pool, than to be missing required information.

5) **Write** the Fact Extractors. In this example it is suggested that a two-step development approach be taken. The first is to develop and test the subordinate fact extractors, perhaps called NamedEntity and Relationship. When these two fact extractors have been developed and tested they can be combined into a relationship fact extractor that looks for patterns of the type "(<NamedEntity>) (<Relationship>) (<NamedEntity>)".

6) **Test** the Fact Extractors. Test each subordinate individually to ensure it is finding all the subordinate facts. Then test the composite to make sure that it is correctly combining the results. It should limit the facts to exclude most false positives from the subordinate facts.

8. Evaluation

8.1 During Development

After scanning a corpus of documents it is often easy to construct a set of patterns that match all the important facts and to then be lulled into a false sense of security about the accuracy of the fact extractor. It is vital to keep some of the document corpus aside for blind testing of the fact extractor, i.e. testing with documents that have not been seen by the developer.

A second area to consider is that during development, changes to rules may be made that cause the fact extractor to stop working correctly on previously considered documents. From time to time during the development process it is necessary to re-examine old documents in the corpus and verify correct behaviour of the fact extractor under development.

When this document was written the fact extractor workbench provided built-in regression testing over previously-analysed documents. It is intended that future development will provide support for on-the-fly quality evaluation as new documents from the corpus are examined.

8.2 Ongoing Evaluation During Use

Fact extractors are unlikely to ever be part of a set-and-forget system. Information sources evolve over time and as fact extractors are normally tailored to artefacts of the information source they will require routine maintenance. In the case of fact extractors used within the FormFiller application this will usually[26] become obvious to the users and they will request updates. In the case of fact extractors used in an automated system this gradual degradation is likely to go unnoticed. It is recommended that from time to time the document stream is randomly sampled and the performance of the fact extractors over the sample documents examined. The frequency and completeness of this evaluation is dependant on the accuracy requirements of the system.

9. Summary

This paper has described several applications of information extraction, and the tradeoffs in requirements for recall, precision, accuracy and speed. It has also described a number of concepts that apply to using DSTO's fact extractor technology and provided a number of examples of how to use each part of a fact extractor. The examples included a number of guidelines and principles that are collected here in

---

[26] FormFiller users may not notice if some facts go undetected.

summary. Following these principles will lead to more effective development of fact extractors.

The general principles for developing effective fact extractors are:

1. Understand the information need before starting development.

2. It is generally not practical to craft a perfect fact extractor so consider the speed and accuracy requirements.

3. Use a corpus of real text.

4. Keep some of the corpus aside to enable proper testing.

5. Copy interesting sentences from the corpus into a single document to facilitate development and testing.

6. Identify parts of the text that are key to the existence of a fact, and write create rules for these sections.

7. For relevant text, which on its own wouldn't signify the presence of a fact, develop appropriate expand and co-reference rules to connect with the key text.

8. Develop and test the create rules first. Then introduce expand and co-reference rules.

9. Develop and test the pattern part of each rule first and then add the actions.

10. Use subordinate fact extractors to simplify complex tasks.

11. Use a list fact extractor at any time if it simplifies the task. Remember that lower case lists do case-insensitive matching.

12. For better execution speed, use a list fact extractor if there are more than about twenty simple items.

13. Avoid using guard expressions to constrain very freely matching patterns. Patterns evaluate faster than guard expressions.

14. Avoid using subordinates that match very frequently (thousands of matches) by themselves but rarely contribute to the composite.

15. Co-reference rules potentially evaluate their guard expression for every fact found so far. Avoid using loose patterns, particularly if large numbers of facts may be found per document.

16. Reuse design principles as well as trying to directly reuse previously developed fact extractors.

17. Collaborate with others on fact extractor development.

18. Plan to maintain fact extractors over time.

10. Further Considerations

The initial intent of the Fact Extractor project was to demonstrate that complex information could be found by combining simple building blocks. While this has been demonstrated, the simple building blocks have not proved to be as reusable as originally anticipated. We expected to be able to build a wide range of completely generic fact extractors that would then be able to be composed together to satisfy a particular need, and the fact extractor workbench was designed with this goal in mind. However it has become apparent that it is unusual to be able to build fact extractors that behave well across a wide range of different document styles and sources. Each author, editor, or intended audience tends to have a specific style, and it is often not possible to identify the significant concepts in different sources of documents in a completely generic manner for non-trivial information needs.

Although reuse of specific fact extractors for a new purpose often proves impractical, aspects of their design may be reused. This includes the choice of composite and subordinate fact extractors; the choice between create, expand and co-reference rules; and the number and granularity of fields. Accordingly, any future redevelopment of the fact extractor workbench will be directed toward making this task easier.

References

[1]   P L Choong "Adaptive Information Extraction: Research and Development Trends", DSTO-CR-0363, Defence Science and Technology Organisation, 2004.

[2]   P Wallis & G Chase "Beyond Keywords: Getting from Text to Information", Proceedings of the 1998 Optimising Open Source Information Conference.

[3]   P Wallis & G Chase "An Information Extraction System", Australasian Natural Language Processing Summer Workshop, February 1997.

[4]   Jeffery E F Friedl, "Mastering Regular Expressions", O'Reilly & Associates Inc, 1997, First Edition, USA.

[5]   Perl Regular Expressions, http://www.perldoc.com/perl5.6/pod/perlre.html Last updated 9 Sept 2001; checked March 2005.

[6]   FormFiller application is part of the Fact Extractor System software distribution available on the Defence Restricted Network from:
       http://sourceforge.dsto.defence.gov.au/projects/factextractors/

[7]   Google ™ is a trademark of Google Technology Inc, http://www.google.com

[8]   J Das, G Chase & S Davis "Fact Extractor System Processing Engine", DSTO-TR-1396, Defence Science and Technology Organisation, March 2003.

[9]   Linguistic Data Consortium Web Site: http://www.ldc.upenn.edu

[10]   ISO8601:2000, *Data elements and interchange formats– Information interchange – Representation of dates and times.* International Organization for Standardization (ISO)

[11]   ISO 3166-1 *English Country Names,* International Organization for Standardization (ISO)

[12] The FXBench application is part of the Fact Extractor System software distribution available from the authors. The application is distributed with online help.

[13] B Williams and N Banks "Normalcy Analysis Toolkit User Guide", DSTO-IP-0031, Defence Science and Technology Organisation, 2002.

[14] S Heath "Performance Analysis and Optimisation of the Fact Extractor System", DSTO-TN-0566, Defence Science and Technology Organisation, 2004.

[15] *Web Naming and Addressing Overview (URIs, URLs, ...)*, World Wide Web Consortium http://www.w3.org/Addressing/   Last updated 2005-02-17, revision 1.56. Created 1993 by Tim Berners-Lee.

Appendix A The Regular Expression Based Pattern Language

Regular expressions are a standard notation for characterising text sequences. Regular expression language is a language used for specifying text search strings [4]. The fact extractor system uses regular expressions to detect patterns in text.

The pattern part of a rule specifies the criteria for identifying facts. The patterns are based on regular expressions with an extension to allow the inclusion of subordinate Fact Extractors. Refer to the Perl 5 manual[5][4] for a full description of Regular Expressions. This section will provide a summary of the common features used in developing Fact Extractors and our extension to support embedded subordinate Fact Extractors.

A.1. Simple Character Patterns

- A single character that matches itself, eg. "a" matches "a".

- "xyz" matches an "x" followed by a "y" followed by a "z".

- A dot "." matches any single character except newline, e.g. "a." matches any two character sequence starting with "a".

- "^" at the beginning of a pattern forces it to match the start of a sentence.

- "$" at the end of a pattern forces it to match the end of a sentence.

- "\b" matches a word boundary.

- "\w" matches a word character (alphanumeric and "_").

- "\W" matches a non-word character.

- "\d" matches a numeric character.

- "\s" matches a white-space character such as a space, tab or newline.

A.2. Specifying Groups

- Parentheses "()" can be used to indicate a pattern group.

- Groups are used to control the aggregation of quantifier operators. Group numbers are used for the FIELD and MATCH action functions, the matchrange field, and in other parts of regular expressions, for example in backward referencing.

A.3. Repetitive Patterns

Repetitive patterns match multi-character sequences, for example:

- The asterisk "*" matches zero or more of the previous group,

- a* matches "", "a", "aa", "aaa", etc.

- abc* matches "ab", "abc", "abcc", etc.

- (abc)* matches "", "abc", "abcabc", etc.

- The plus sign "+" matches one or more of the previous group,

    - a+ matches "a", "aa", "aaa", etc., but not "".

    - abc+ matches "abc", "abcc", etc.

    - (abc)+ matches "abc", "abcabc", etc.

    - "\w+" matches a sequence of word characters.

- The question mark "?" matches zero or one of the previous group,

    - a? matches "" and "a".

    - abc? matches "ab" and "abc".

    - (abc)? matches "" and "abc".

- Braces "{n[,m]}" match at least n but not more than m times,

    - a{3} matches only "aaa".

    - a{3,5} matches "aaa", "aaaa", "aaaaa".

By default repetitive groupings are greedy; they match as many characters as possible. To reverse this preference, follow the quantifier with a "?".

A.4. Alternation

- a list of characters or character ranges enclosed in square brackets matches any one of them,

    - [ab] matches "a" or "b".

    - [3456789] matches any single digit in the range 3 to 9.

    - [3-9] is shorthand for the above.

    - [a-zA-Z0-9] matches any single letter or digit.

- A "^" indicates not in the set, for example [^0-9] matches any character which is not a digit.

- The or "|" matches a set of alternatives,

    - "abc|def" matches "abc" or "def".

    - AM|PM will match either "AM" or "PM". This could also be written as [AP]M.

A.5. Extracting Data

Parentheses make sections of the matched text available to the action part of a pattern-action rule. For example, a pattern such as "`(Mr|Mrs|Dr|Ms) ([A-Z][a-z]*)`" will match a title and a surname such as "Mr Smith".

The parenthesized groups are made available to Action functions (see Appendix B). For the above example, `MATCH(0)` accesses the whole matching string, `MATCH(1)` accesses the title and `MATCH(2)` accesses the surname.

A.6. Embedding Another Fact Extractor In A Pattern

Much of the power of the DSTO Fact Extractor System comes from the ability to embed a previously developed Fact Extractor in the pattern part of a pattern-action rule. For example, a **Date** Fact Extractor may wish to embed a **Month** Fact Extractor in one or more of its pattern-action rules. Embedded Fact Extractors are enclosed in angle brackets. For example, a simple **Date** Fact Extractor which matches dates of the form: "dd mmm yyyy" where <Month> is assumed to match three letter abbreviations of month could be represented by the modified Regular Expression:

`\b([0-3]?[0-9]) (<Month>),? ([12][0-9][0-9][0-9])`

The sets of parentheses in the above example will enable the day, month and year to be used in the action part of the pattern-action rule.

To reiterate, embedded subordinate Fact Extractors are enclosed in angle brackets. Regular Expression Language meta-characters cannot be used on embedded subordinate Fact Extractors, for example, "`<numberFX>?`" is not valid.

Appendix B   The Action Language

The table below describes functions available in the FX Expression Language Functions Suite. These functions can be used in guard expressions to constrain a match, and in action expressions to assign values to Fact Extractor fields. When a running Fact Extractor invokes an FX Expression Language Function with invalid argument(s) in a rule Action or Guard Expression, an error message is generated and reported.

The Fact Extractor System is not limited to the set of functions described below. Any Java static function which takes only String parameters and returns a String can be invoked simply by using the fully-qualified class and method name.

ADD(arg1, ...)	returns the sum of its arguments
AND(arg1, ...)	returns the logical AND of the arguments
ARCCOS(arg)	returns the arc cosine of an angle (in radians)
ARCSIN(arg)	returns the arc sine of an angle (in radians)
CACHE(fxname, fieldname [, default])	searches the cache of recent facts for fxname, and returns the value of fieldname (or default if not found)
CONCAT(arg1, ...)	returns the concatenation of its arguments
COS(arg)	returns the cosine of an angle (in radians)
DBLOOKUP(datasource, sqlstatement [, userid, password])	returns a value from a database
DIVIDE(dividend, divisor)	returns dividend/divisor
EQUAL(arg1, arg2)	returns TRUE if they are the same, FALSE otherwise
EXISTS(value)	checks to see if the value is not empty
FIELD(matchnum, fieldname)	returns the value of the named field of the subordinate FX at matchnum

FORMATNUMBER(number, format)	returns number in the specified format. See the online help for format specifications
GREATERTHAN(arg1, arg2)	returns TRUE if arg1 > arg2, FALSE otherwise
IF(condition, iftrue, iffalse)	if the condition is true, returns the result of evaluating iftrue; otherwise, returns the result of evaluating iffalse.
LESSTHAN(arg1, arg2)	returns TRUE if arg1 < arg2, otherwise FALSE
MATCH(n)	returns the nth parenthesised group from the pattern. Groups are numbered in the order that their left parenthesis appears in the pattern. MATCH(0) is the whole pattern
MULTIPLY(arg1, ...)	returns the product of its arguments
NOT(arg1)	returns the logical opposite of the argument
OR(arg1, ...)	returns the logical OR of the arguments
PI()	returns the nearest possible representation to PI, the ratio of the circumference of a circle to its diameter
PROPERTY(propertyname)	not implemented yet, but will return the value of the named property of the sentence
SIN(arg)	returns the sine of an angle (in radians)
STRLENGTH(string)	returns the length of the string argument
SUB(s, from, to)	returns the substring of s defined by from and to
SUBTRACT(arg1, arg2)	returns the difference between the two numeric arguments (arg1 - arg2)
TOUPPERCASE(string)	returns the given string argument in lower case

TOLOWERCASE(string)	returns the given string argument in lower case
VALUE(name)	Inside an action expression returns the value of the field "name" of the fact currently being populated by the rule.
	It is not valid to use this action inside the guard expression of a create rule as the values haven't been set up yet.
	Inside the guard expression for a co-reference rule it returns the value of a field of a previous fact and may be used to aid in determining if that fact is really a co-reference.
	Inside the guard expression of an expand rule it returns the value of a field from the fact that is being expanded.

DISTRIBUTION LIST

Towards Developing Effective Fact Extractors

Greg Chase, Jyotsna Das and Scott Davis

AUSTRALIA

DEFENCE ORGANISATION

	No. of copies
Task Sponsor:	
ASCCR, Defence Intelligence Organisation (R4-3-201)	Doc Data Sheet
DINTCAP, Defence Intelligence Organisation (R4-3-181)	1
ADIIE, Defence Intelligence Organisation (R4-3-182)	1
S&T Program	
Chief Defence Scientist	
FAS Science Policy	shared copy
AS Science Corporate Management	
Director General Science Policy Development	
Counsellor Defence Science, London	Doc Data Sheet
Counsellor Defence Science, Washington	Doc Data Sheet
Scientific Adviser to MRDC, Thailand	Doc Data Sheet
Scientific Adviser Joint	1
Navy Scientific Adviser	Doc Data Sheet
Scientific Adviser - Army	Doc Data Sheet
Air Force Scientific Adviser	Doc Data Sheet
Scientific Adviser to the DMO	Doc Data Sheet
Information Sciences Laboratory	
Chief Command & Control Division	Doc Data Sheet
Research Leader Command Decision Environments Branch	1
Research Leader Information Enterprises Branch	Doc Data Sheet
Research Leader Joint Command Analysis Branch	Doc Data Sheet
Research Leader Intelligence Information Branch	1
Head Human Systems Integration	Doc Data Sheet
Head Information Exploitation	Doc Data Sheet
Head Effects-Based Modelling and Analysis	Doc Data Sheet
Head Information Systems	Doc Data Sheet
Head Distributed Enterprises	Doc Data Sheet
Head Joint Operations Analysis and Support	Doc Data Sheet
Head Command Concepts and Architectures	Doc Data Sheet
Head Command Process Integration and Analysis	Doc Data Sheet
Head Intelligence Analysis	1

Greg Chase (Author)	1
Jyotsna Das (Author)	2
Scott Davis (Author)	1
Dr P L Choong, IA Group C2D	1
Dr D Parker, IA Group C2D	1
Dr T Pattison, C2D	1
Mr B Dennis, C2AST C2D (Potts Point NSW)	1
Publications and Publicity Officer, C2D/EOC2D	1 shared copy

Science Policy Division

Mr David Clayton, A/DT2CO, DSTO Edinburgh	1

DSTO Library and Archives

Library Edinburgh	1 + Doc Data Sheet
Defence Archives	1

Capability Development Group

Director General Maritime Development	Doc Data Sheet
Director General Land Development	Doc Data Sheet
Director General Integrated Capability Development	Doc Data Sheet
Director General Capability and Plans	Doc Data Sheet
Assistant Secretary Investment Analysis	Doc Data Sheet
Director General Capability Plans and Programming	Doc Data Sheet
Director General Australian Defence Simulation Office	Doc Data Sheet

Chief Information Officer Group

Director General Australian Defence Simulation Office	Doc Data Sheet
Director General Information Policy and Plans	Doc Data Sheet
AS Information Strategy and Futures	Doc Data Sheet
AS Information Architecture and Management	Doc Data Sheet
Director General Information Services	Doc Data Sheet

Strategy Group

Director General Military Strategy	Doc Data Sheet
Assistant Secretary Governance and Counter-Proliferation	Doc Data Sheet

Navy

Maritime Operational Analysis Centre, Building 89/90	Doc Data Sht & Dist
Garden Island Sydney NSW	List (shared)
Deputy Director (Operations)	
Deputy Director (Analysis)	
Director General Navy Capability, Performance and Plans, Navy Headquarters	Doc Data Sheet
Director General Navy Strategic Policy and Futures, Navy Headquarters	Doc Data Sheet

Air Force

SO (Science) - Headquarters Air Combat Group, RAAF Base,
Williamtown NSW 2314 — Doc Data Sht & Exec Summ

Army

ABCA National Standardisation Officer

Land Warfare Development Sector, Puckapunyal — e-mailed Doc Data Sheet

SO (Science) – Land Headquarters (LHQ)
Victoria Barracks NSW — Doc Data Sht & Exec Summ

SO (Science), Deployable Joint Force
Headquarters (DJFHQ) (L), Enoggera QLD — Doc Data Sheet

Joint Operations Command

Director General Joint Operations — Doc Data Sheet
Chief of Staff Headquarters Joint Operations Command — Doc Data Sheet
Commandant ADF Warfare Centre — Doc Data Sheet
Director General Strategic Logistics — Doc Data Sheet
COS Australian Defence College — Doc Data Sheet

Intelligence and Security Program

AS Concepts, Capability and Resources — 1
DGSTA, DIO — 1
COMD JOIC, Joint Operational Intelligence Centre — Doc Data Sheet
Director, Business Information Management,
Defence Intelligence Organisation — 1
PDJISS, Joint Intelligence Support System — Doc Data Sheet
PBD, Defence Signals Directorate — Doc Data Sheet
J2HQAST — Doc Data Sheet
Manager, Information Centre, Defence Intelligence
Organisation — 1 (PDF version)
Assistant Secretary Corporate, Defence Imagery and
Geospatial Organisation — Doc Data Sheet
Assistant Secretary Capability Provisioning — Doc Data Sheet
Assistant Secretary Capability and Systems — Doc Data Sheet

Defence Materiel Organisation

Deputy CEO — Doc Data Sheet
Head Aerospace Systems Division — Doc Data Sheet
Head Maritime Systems Division — Doc Data Sheet
Head Electronic and Weapon Systems Division — Doc Data Sheet
Program Manager Air Warfare Destroyer — Doc Data Sheet

Defence Libraries

Library Manager, DLS-Canberra — Doc Data Sheet

OTHER ORGANISATIONS
    National Library of Australia   1
    NASA (Canberra)   1
    State Library of South Australia   1

UNIVERSITIES AND COLLEGES
    Australian Defence Force Academy
        Library   1
        Head of Aerospace and Mechanical Engineering   1
    Serials Section (M list), Deakin University Library,
        Geelong, VIC   1
    Hargrave Library, Monash University   Doc Data Sheet
    Librarian, Flinders University   1

<div align="center">

OUTSIDE AUSTRALIA

</div>

INTERNATIONAL DEFENCE INFORMATION CENTRES
    US Defense Technical Information Center   1 PDF
    UK Dstl Knowledge Services   1 PDF
    Canada Defence Research Directorate R&D Knowledge
        & Information Management (DRDKIM)   1
    NZ Defence Information Centre   1

ABSTRACTING AND INFORMATION ORGANISATIONS
    Library, Chemical Abstracts Reference Service   1
    Engineering Societies Library, US   1
    Materials Information, Cambridge Scientific Abstracts, US   1
    Documents Librarian, The Center for Research Libraries, US   1

SPARES   5

Total number of copies: Printed 40   +   PDF 3   =   43

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION DOCUMENT CONTROL DATA	1. PRIVACY MARKING/CAVEAT (OF DOCUMENT)

2. TITLE Towards Developing Effective Fact Extractors	3. SECURITY CLASSIFICATION (FOR UNCLASSIFIED REPORTS THAT ARE LIMITED RELEASE USE (L) NEXT TO DOCUMENT CLASSIFICATION) Document (U) Title (U) Abstract (U)

4. AUTHOR(S) Greg Chase, Jyotsna Das and Scott Davis	5. CORPORATE AUTHOR DSTO *Defence Science and Technology Organisation* PO Box 1500 Edinburgh South Australia 5111 Australia

6a. DSTO NUMBER DSTO-TR-1729	6b. AR NUMBER AR-013-418	6c. TYPE OF REPORT Technical Report	7. DOCUMENT DATE June 2005

8. FILE NUMBER 2004/1065566/1	9. TASK NUMBER INT 02/290	10. TASK SPONSOR ASCCR DIO DINTCAP DIO ADIIE DIO	11. NO. OF PAGES 56	12. NO. OF REFERENCES 14

13. URL on the World Wide Web http://www.dsto.defence.gov.au/corporate/reports/DSTO-TR-1729.pdf	14. RELEASE AUTHORITY Chief, Command and Control Division

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

Approved for public release

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SA 5111

16. DELIBERATE ANNOUNCEMENT

No Limitations

17. CITATION IN OTHER DOCUMENTS	Yes

18. DEFTEST DESCRIPTORS

Intelligence
Text processing
Information extraction

19. ABSTRACT

DSTO has a program of research into automated text processing. Part of this research has led to the development of a prototype information extraction system known as the DSTO Fact Extractor System. This system can be used to extract interesting information from free text documents. Part of applying the DSTO technology involves a skilled user developing a set of one or more fact extractors that control the behaviour of the information extraction engine. These fact extractors are developed with the aid of an integrated development environment known as the Fact Extractor Workbench. This report uses a range of examples to discuss the issues that must be considered when developing fact extractors.